你也能看得懂的Python算法书

王硕 董文馨 张舒行 张洁◎编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

编程的核心是算法，学习算法不仅能教会你解决问题的方法，而且还能为你今后的发展提供一种可能。

本书面向算法初学者，首先介绍当下流程的编程语言 Python，详细讲解 Python 语言中的变量和循序、分支、循环三大结构，以及列表和函数的使用，为之后学习算法打好基础。然后以通俗易懂的语言讲解双指针、哈希、深度优先、广度优先、回溯、贪心、动态规划和最短路径等经典算法。

本书适合有一定编程基础的算法爱好者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

你也能看得懂的 Python 算法书/王硕等编著.—北京：电子工业出版社，2018.11
ISBN 978-7-121-35255-3

Ⅰ. ①你… Ⅱ. ①王… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2018）第 240979 号

策划编辑：张月萍
责任编辑：刘　舫
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：16.25　　字数：352 千字
版　　次：2018 年 11 月第 1 版
印　　次：2018 年 11 月第 1 次印刷
印　　数：3500 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

为什么要写这本书

算法是编程的核心，就像一台计算机的 CPU，算法的好坏决定了一个系统的效率高低。

许多人认为学习编程就是学习最新的编程语言、技术和框架，其实计算机算法更重要。计算机语言和技术日新月异，但万变不离其宗的是算法。修炼好算法这门“内功”，再辅以新技术这些“招式”，才能独霸“武林”。这也是为什么像 Google 和 Facebook 这类大公司在面试中主要会考查算法问题的原因。

目前图书市场上关于算法的图书不少，经典的如《算法导论》。但是大多数算法图书太学术、太复杂，对于初学者来说门槛太高。学习算法本身就不是一件容易的事情，再加上复杂的场景和数学理论，会让算法的学习曲线更陡。因此本书的作者们就萌生了写一本让大家都能看懂的算法书的想法，以生动的语言把算法的思想过程写出来，让学习算法不再那么枯燥。

在本书的创作过程中，王硕老师迎来了人生的第一个宝宝小朗朗。于是几位作者就有了一个心愿，希望青少年朋友也学会算法，因此，在描述算法问题时，尽量用简单、通俗的形式表述，使青少年朋友也能看懂，也希望如此这般能够增加读者的学习兴趣。既然是学习算法，免不了要写代码，对于编程语言的选择，我们选择了 Python 这门简单易懂的语言作为本书的编程语言。对于初学编程的人来说，Python 可以缩短学习编程语言的时间和降低学习编程的难度。

学习算法不易，且行且珍惜。

本书有何特色

1. 以孩子的口吻，生动形象地讲解算法，提高趣味性

为了便于读者理解本书内容，提高学习效率，大部分问题都以孩子的口吻来讲解，以解决小朗朗的实际问题作为出发点引出问题，增加了趣味性和可读性。

2. 涵盖核心算法知识点

本书涵盖双指针问题、哈希、深度优先遍历、广度优先遍历、回溯、贪心、动态规划、最短路径问题、分治等 9 大算法，帮助读者全面掌握核心算法的知识点。

3. 以 Python 语言作为载体，降低学习难度

抛弃其他复杂的编程语言，本书采用简单的编程语言 Python 作为算法的载体，并在第 1 章介绍了 Python 语言的语法。

4. 选择经典算法的经典问题，有较高的通用性

本书在简单介绍 Python 编程语言以后，选择了 9 大经典算法，重点讲解算法原理，并选择经典问题进行有针对性的练习。

5. 提供完善的技术支持和售后服务

本书提供了专门的邮箱供读者咨询：317977682@qq.com。读者在阅读本书的过程中有任何疑问都可以通过该邮箱获得帮助。

本书内容及知识体系

第 1 章　编程基础

掌握一门编程语言是学习算法的基础。学习编程语言是为了与计算机“交流”，只有正确的格式才能被计算机成功识别。学习编程语言的起点就是了解这门语言的语法。本书使用 Python 语言进行算法讲解，本章主要讲解 Python 3 的编程语法。

第 2 章　双指针问题

“指针”是编程语言中的一个对象，它存储着一个内存空间的地址，计算机可以通过这个地址找到变量的值。也就是说，这个特定的地址指向这个特定的值。指针最大的优点在于它可以有效利用零碎的内存空间。

第 3 章　哈希算法

哈希算法又称散列函数算法，是一种查找算法，简单来说，就是把一些复杂的数据，通过某种函数映射关系，映射成更加易于查找的方式。但是这种映射关系有可能会发生多个关键字映射到同一地址的现象，我们称之为冲突。在这种特殊情况下，需要对关键字进行第二次或更多次的处理，在其他的大多数情况下，哈希算法可以实现在常数时间内存储和查找这些关键字。

第 4 章　深度优先遍历算法

深度优先遍历算法是经典的图论算法，它的搜索逻辑就和它的名字一样，只要有可能，就尽量深入搜索，直到找到答案，或者尝试了所有可能后确定没有解。

第 5 章　广度优先遍历算法

广度优先遍历算法与深度优先遍历算法类似，也是查询的方法之一，它从某个状态出发查询可以到达的所有状态。但不同于深度优先遍历，广度优先遍历算法总是先去查询距离初始状态最近的状态。

第 6 章　回溯算法

回溯算法可以被看作走迷宫，因为我们不知道出口在哪里，所以只能不断地深入，尝试不同的路线。一旦找到了出口便可以回溯到起点，辨清路线。

第 7 章　贪心算法

贪心算法就是遵循某种既定原则，不断地选取当前条件下最优的选择来构造每个子步骤的解决方案，直到获得问题最终的解。即在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，它所做的仅是在某种意义上的局部最优解。

第 8 章　动态规划算法

动态规划算法将待求解问题拆分成一系列相互交叠的子问题，通过递推关系定义各子问题的求解策略，并随时记录子问题的解，最终获得原始问题的解，避免了对交叠子问题的重复求解。

第 9 章　最短路径问题

把地点看成节点，把道路看成边，整个地图就可看成一个加权图。计算加权图中两点间的最短路径是编程的一个重要问题，这一章我们会用以下几个算法解决这个问题：迪可斯特朗算法、Floyd 算法、A*算法。

第 10 章　分治算法

分治算法的核心思想是把一个规模很大的问题化简为多个规模较小的问题，这些子问题虽然独立而不同，但是问题的本质是一致的，从而达到分而治之的目的。

适合阅读本书的读者

- 需要全面学习算法的人员
- 需要学习 Python 的程序员
- 对编程算法感兴趣的人员
- 希望提高算法水平的程序员
- 专业培训机构的学员

阅读本书的建议

- 没有 Python 基础的读者，建议从第 1 章开始顺次阅读并演练每个实例。
- 有一定 Python 基础的读者，可以根据实际情况有重点地选择阅读各个模块和项目案例。
- 对于每个模块和案例，先自己思考一下实现的思路，然后再阅读，学习效果会更好。这样理解起来也会更加容易、更加深刻。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35255

目录

第 1 章　编程基础 1

1.1　变量 1
1.1.1　输出和输入 2
1.1.2　简单变量类型 3
1.1.3　数学计算 6
1.1.4　位运算 7
1.1.5　使用字符串 11
1.2　三大结构 15
1.2.1　循序结构 15
1.2.2　分支结构 16
1.2.3　条件判断 18
1.2.4　应用分支结构 20
1.2.5　循环结构 21
1.2.6　continue 和 break 23
1.2.7　应用循环结构 24
1.2.8　结构的嵌套 26
1.3　列表 27
1.3.1　定义列表 27

1.3.2 对元素进行操作 28
1.3.3 列表的顺序 31
1.3.4 列表内置函数 33
1.3.5 截取和拼接列表 36
1.3.6 字符串、元组和列表 38
1.3.7 用循环遍历列表 40
1.3.8 字典简介 41
1.4 函数 43
1.4.1 定义子函数 43
1.4.2 主函数 44
1.4.3 调用函数 45
1.4.4 全局变量 47
1.4.5 函数的运用 48

第 2 章 双指针问题 53

2.1 数组合并 53
2.1.1 合并有序数组 53
2.1.2 最终代码 56
2.2 二分查找 56
2.2.1 什么是二分查找 57
2.2.2 问题求解 58
2.2.3 最终代码 60
2.3 链表 60
2.3.1 什么是单链表 60
2.3.2 建立单链表 61
2.3.3 建立双链表 63
2.3.4 双向输出双链表 65
2.3.5 向单链表中添加元素 66
2.3.6 向双链表中添加元素 69
2.3.7 删除列表中的元素 71

第 3 章　哈希算法 75

3.1　什么是哈希 75
3.2　两个数的和 78
3.2.1　问题求解 1 78
3.2.2　解法 1 的最终代码 80
3.2.3　问题求解 2 81
3.2.4　解法 2 的最终代码 82
3.3　单词模式匹配 82
3.3.1　问题求解 83
3.3.2　最终代码 85
3.4　猜词游戏 85
3.4.1　问题求解 87
3.4.2　最终代码 88
3.5　神奇的词根 89
3.5.1　问题求解 90
3.5.2　最终代码 92

第 4 章　深度优先遍历算法 93

4.1　什么是深度优先遍历 93
4.2　二叉树 95
4.2.1　二叉树的类型 95
4.2.2　二叉树的相关术语 96
4.2.3　二叉树的节点代码 97
4.2.4　二叉树的遍历顺序 97
4.2.5　深度优先遍历与广度优先遍历 97
4.3　怎么抓住小偷 98
4.3.1　解题思路 98
4.3.2　从思路到代码 102
4.4　二叉树中的最大路径和 102
4.4.1　解题思路 103

4.4.2 完整代码 112
4.5 最大的岛屿 113
4.5.1 解题思路 113
4.5.2 完整代码 116

第 5 章 广度优先遍历算法 118

5.1 什么是广度优先遍历 118
5.2 选课的智慧 120
5.2.1 广度优先遍历 121
5.2.2 问题求解 122
5.2.3 最终代码 124
5.3 寻找制高点 125
5.3.1 问题求解 126
5.3.2 集合 129
5.3.3 最终代码 130
5.4 合法的括号 131
5.4.1 问题求解 131
5.4.2 最终代码 135
5.5 树的右侧 136
5.5.1 问题求解 136
5.5.2 最终代码 139

第 6 章 回溯算法 141

6.1 什么是回溯 141
6.2 遍历所有排序方式 142
6.2.1 问题求解 142
6.2.2 最终代码 144
6.3 经典问题的组合 147
6.3.1 问题求解 147
6.3.2 最终代码 149
6.4 查找单词问题 151

6.4.1 问题求解 152
6.4.2 最终代码 155
6.5 八皇后问题 157
6.5.1 问题求解 158
6.5.2 最终代码 160
6.6 教你解数独 164
6.6.1 问题求解 165
6.6.2 最终代码 168

第 7 章 贪心算法 172

7.1 硬币找零问题 173
7.1.1 问题描述 173
7.1.2 最终代码 175
7.2 活动安排问题 175
7.2.1 问题描述 176
7.2.2 最终代码 177
7.3 哈夫曼编码 178
7.3.1 问题描述 178
7.3.2 哈夫曼树 179
7.3.3 贪心选择性质 181
7.3.4 最优子结构性质 182
7.3.5 最终代码 183

第 8 章 动态规划算法 185

8.1 爬楼梯问题 185
8.1.1 问题描述 186
8.1.2 最终代码 188
8.2 矿工挖矿问题 189
8.2.1 问题描述 189
8.2.2 最终代码 195
8.3 背包问题 195

8.3.1 问题描述 195
8.3.2 问题实例 196
8.3.3 最终代码 201
8.4 最长递归子序列问题 202
8.4.1 问题描述 202
8.4.2 改进算法 204
8.4.3 最终代码 205

第 9 章 最短路径问题 207

9.1 迪可斯特朗算法 207
9.1.1 术语释义 208
9.1.2 问题示例：最短公交线路 208
9.1.3 图与节点的定义 209
9.1.4 把图用代码“画”出来 210
9.1.5 算法核心：两个节点集合 210
9.1.6 算法核心：循环 210
9.1.7 输出路线 211
9.1.8 通过示例理解算法 211
9.1.9 完整代码展示 214
9.2 Floyd 算法 216
9.2.1 算法核心：两个矩阵 216
9.2.2 算法核心：通过中介点缩短距离 217
9.2.3 通过示例理解算法 218
9.2.4 完整代码 222
9.3 A*算法 223
9.3.1 算法核心：迪可斯特朗算法 223
9.3.2 算法核心：预估函数 224
9.3.3 算法核心：选择预估函数 226
9.3.4 A*算法的兄弟们 226

第 10章　分治算法 227

10.1　什么是分治.....227
10.2　归并排序.....228
10.2.1　递归法与迭代法.....228
10.2.2　递归法描述.....229
10.2.3　迭代法描述.....232
10.2.4　最终代码.....233
10.3　连续子列表的最大和.....235
10.3.1　解题思路.....235
10.3.2　最终代码.....237
10.4　几何问题之凸包.....238
10.4.1　问题求解.....238
10.4.2　最终代码.....240
10.5　数学问题之多项式乘法.....242
10.5.1　问题求解.....242
10.5.2　最终代码.....245

第 1 章
编程基础

掌握一门编程语言是学习算法的基础。学习编程语言是为了与计算机“交流”，只有正确的格式才能被计算机成功识别。学习编程语言的起点就是了解这门语言的语法。本书使用 Python 语言进行算法讲解，本章主要讲解 Python 3 的编程语法，涉及的主要内容有如下几项。

- 变量：变量的类型和操作。
- 三大结构：循序、分支和循环。
- 数组：数组的概念和使用。
- 函数：函数的定义与调用。

1.1 变量

变量在计算机语言中，可以用于存储不同的计算结果和不同类型的数据，或表示一个数值。变量是可变的，也就是说，它可以被重复赋值。

在认识变量之前，我们先来了解输入和输出，这样才能对程序输入数据或者看到运行后的结果。

1.1.1 输出和输入

我们以非常经典的一项任务入手，在屏幕上输出一行文字：Hello World!

在 Python 3 中，实现这个任务只需要一行代码（Python 中“#”后方的文字为注释，不影响程序运行，详细信息见 1.1.5 节最后）。

```
01    print("Hello World!")                  #注意：编写程序时一定要用英文输入法！
```

运行程序，输出结果为：

```
    Hello World!
```

在这一行程序中，print()是用于输出的函数，“Hello World!”是输出的数据。其中双引号代表数据是一个字符串。

我们再来看如何输入数据。如果想输入一个数字，把它除以 2 然后打印出来，该怎么做呢？下面的程序就实现了这个功能。

```
01    num = int(input("Enter a number:"))    #输入一个数字，用变量 num 存储
02    num = num/2                            #把 num 除以 2
03    print(num)                             #输出 num 的值
```

运行程序，程序输出如下：

```
Enter a number:
```

此时可以输入任意整数。我们输入 6，程序输出 3 后结束。

在程序的第一行中，新建了变量 num 来存储输入的数据。因为这个变量是用于存储一个数的，所以把它命名为 num，这样比较直观易懂。只要遵守变量的命名法则（参见本小节末尾），那么可以随意为变量取名。

在第一行中，input()函数用于输入数据，括号内的字符串是提示语，就是在输入数据之前输出的那一段字符串。如果括号内没有内容（input()），则不输出任何文字，用户直接输入数据即可。而 input()函数外面套着的 int()函数则是用于把数据转换为 int 类型的函数。Input()函数返回的值默认为字符串类型（string），字符串不可以进行数学运算。所以，我们使用 int()函数先把字符串转换为整型（int），再进行赋值、运算并输出。数据类型的知识会在下一小节介绍。

程序中的“=”用于给变量赋值，不是相等的意思。“=”左边必须是一个可以被赋值的变量，“=”右边可以是一个常量、变量或函数的返回值等。赋值语句运行结束后，“=”左边的变量的值更新为“=”右边的值。与其他编程语言不同，由于 Python 中变量的类型不固定，所以被赋值的变量的数据类型同样可以被更新，如下面的程序所示。

```
variable = 0                    #给变量赋值 0，数据类型为整型（int）
variable = "abcd"               #再给变量赋值"abcd"，数据类型为字符串（string）
print(variable)                 #输出变量
```

运行这段程序，输出结果为 abcd。

在这一小节中我们了解了输入和输出，也对变量的概念有了初步认识。为了避免程序出错，以下是变量命名的几项法则：

- 变量名中只能出现字母、数字和下画线，且第一个字符不能是数字。比如，可以将变量命名为 name_6，但不能将变量命名为 6_name。
- 不要把在 Python 中有特殊用途的单词作为变量名，如函数名。比如，目前我们接触过的 print 函数，如果把 print 作为一个变量名称，运行程序时又用到 print 函数，编译器就会报错。

  下面就是一个例子：

```
print = 0
print(print)                #print 函数无法被调用，print 被系统认为是变量名称了
```

- 一个变量名对应一个变量，不会有两个变量有相同的变量名。变量的名字最好和变量的功能相对应，比如一个存储平均数的变量，将它命名为 average 比命名为 abcd 要直观，这样不容易记错变量名称而导致程序出错。

1.1.2　简单变量类型

Python 语言中有 5 种标准变量类型：数字、字符串、列表、元组和字典，其中数字和字符串是比较简单的，本小节主要介绍这两种变量类型。

首先，我们来仔细了解一下变量的概念。每个变量都有一个独特的变量名，我们通过这个独特的变量名来调用这个变量的值。就像图 1.1 中的气球和牌子一样，人够不着气球，但是可以通过扯动牌子来拿到气球。每个牌子都对应一个单独的气球。牌子就相当于变量名，而气球就相当于变量存储的值。

现在我们知道了变量的概念，再来分类型介绍变量。

数字类型的变量是最常用的一种。数字类型还可以再细分，其中主要的两种类型为整型（int）和浮点型（float）。整型变量只能存储整数，浮点型变量则可以存储小数。

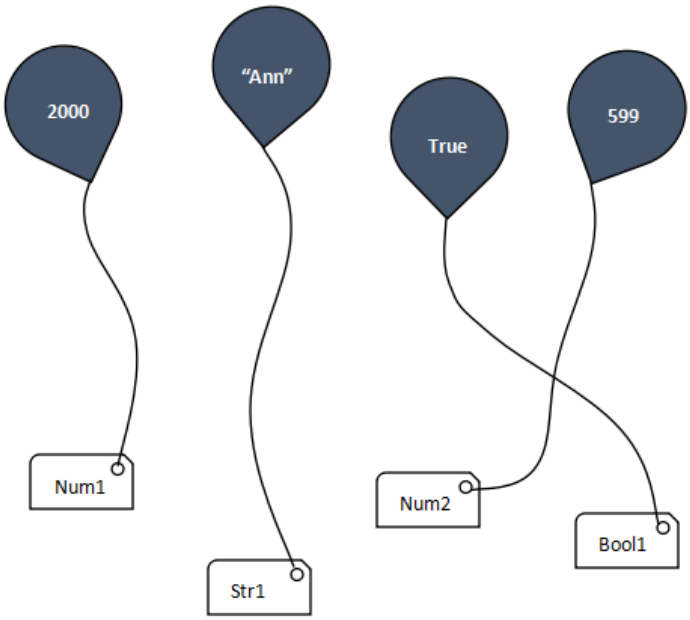


图1.1　变量与变量名

例如：

```
01    num1 = 10
02    num2 = 10.0
```

在这两行程序中，num1 表示的值是整数 10，num2 表示的值是小数 10.0。尽管这两个变量所表示的数的大小是相等的，但它们并不是同一种类型的变量。不过，它们的类型是可以相互转换的。

```
01    num1 = 10
02    num2 = 10.99
03    num1 = float(num1)
04    num2 = int(num2)
05    print(num1,num2)
```

运行程序，输出结果为：

10.0 10

其中，10.0 是输出时 num1 的值，10 是输出时 num2 的值。在程序中，float 函数的作用是把整型变量转换为浮点型变量，而 int 函数的作用是把浮点型变量转换为整型变量。如果你足够细心，会发现原本值为 10.99 的 num2 在转换为整型之后变为了 10，它的值改变了。这是因为，

在 Python 中，不存在四舍五入，取整指的是丢弃原值小数点后的部分。

这段程序中还涉及了关于 print 函数的新语法。我们使用 print(num1,num2)，在同一行输出两个值，中间用空格隔开。

这就好比用一个盒子搬运东西，本来盒子中有多个格子，每个格子都可以装一件物品。使用 print(num1)加上 print(num2)就是每次只用其中的一个格子来搬运东西，所以需要分成两次搬运；print(num1,num2)就是同时用两个格子装物品，一次搬运就可以完成。print 函数就是一个有无数格子的箱子，只要需要就可以使用，如图 1.2 所示。

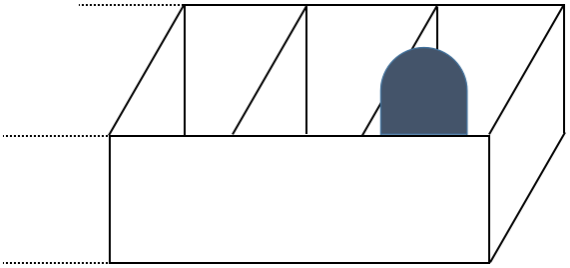


图1.2　print函数与盒子

讲完了新语法，我们再来看看用旧语法输出两个变量时的情况。如果采用分两次 print 的方法来输出，结果会是两个数分别在两行：

```
......
05    print(num1)
06    print(num2)
```

运行程序，输出结果为：

```
10.0
10
```

第二种常用的数据类型是字符串，字符串就是一串字符。在 Python 中，两个单引号或者两个双引号之间的一串字符就是字符串。需要注意的是，如果你使用单引号来表示字符串，那么字符串中不能包含撇号。

```
01    text1 = "I'm a programmer."
02    text2 = 'I'm a programmer.'
```

以上两种字符串定义方式中，第二种是不可行的。系统会认为前单引号和撇号中间的文本 I 为需要定义的字符串，后面字符则会被判定为非法字符，从而导致程序出错。

如果换一个思路，按照下面的方法来定义，就是可行的了。

```
01   text1 = "I'm a programmer."
02   text2 = 'I am a programmer.'
```

1.1.3 数学计算

在 Python 中，可以对数字进行加减乘除运算。其中，加对应“+”，减对应“-”，乘对应“*”，除对应“/”。

以下是 Python 3 中整数运算的程序：

```
01   print(5+2)
02   print(5-2)
03   print(5*2)
04   print(5/2)
```

运行程序，输出结果为：

```
7
3
10
2.5
```

值得注意的是，在 Python 3 中，如果整数相除的结果不是整数，Python 会把运算的结果存储为浮点型的变量。

我们来看一看浮点数的运算：

```
01   print(0.3+0.2)
02   print(0.3-0.2)
03   print(0.3*0.2)
04   print(0.3/0.2)
```

运行程序，输出结果为：

```
0.5
0.09999999999999998
0.06
1.4999999999999998
```

在浮点数的运算中有时会出现第二个和第四个运算结果这样的情况。计算机内部表示数字的方式使得有时表示运算结果出现困难，最后输出一个十分近似真实结果的值。所有语言都存在这样的问题，不必担心。

除了简单的加减乘除运算，Python 中还有许多内置函数，可以对数字进行更复杂的运算。调用这些函数比自己写出实现这些功能的代码要快得多，从而可以大大提高程序的效率，也可

以提高程序的运行速度。下面我们来了解几个常用的数学函数。

pow 函数是十分常见的一个函数，通常用于求一个数的 *n* 次方。

```
01    print(pow(4,2))
02    print(pow(4,2,3))
```

运行这段程序，输出结果为：

```
16
1
```

在这段程序中，我们见到了 pow 函数的两种使用方法。pow 函数的使用格式为 pow(x,y,z)，如果括号内没有填写 z，也就是只输入了两个数的话，pow 函数的返回值为 x 的 y 次方。如果输入了 z，那么 pow 的返回值就是 x 的 y 次方除以 z 的余数，即 x**y%z。在 Python 中，符号“%”是取余的意思，“a%b”就是 a 除以 b 的余数。

在这段程序中，第一行输出的是 4 的 2 次方，也就是 16；第二行输出的是 4 的 2 次方除以 3 的余数，也就是 16/3 的余数 1。

接下来要了解的函数是 abs 函数和 sqrt 函数。

```
01    import math
02    num = -4
03    num = abs(num)
04    print(math.sqrt(num))
```

运行程序，输出结果为：

```
2.0
```

这段程序中使用了用于求绝对值的 abs 函数和用于求平方根的 sqrt 函数。我们知道，负数是没有实数根的，所以在开方之前先用 abs 函数把 num 变成非负数，再把 abs 函数返回的值赋给 num，从而达到更新 num 的值的效果。第 3 行运行结束时 num 的值为 4。

第 4 行使用 sqrt 函数来求 num 的平方根。由于 sqrt 函数不处于默认模块中，所以要先导入包含 sqrt 的 math 模块，使用 import 语句来导入。调用 sqrt 函数时，函数名前面要加上“math.”，表示它是 math 模块内的函数，这样计算机才能找到它。sqrt 函数的返回值是浮点型。

Python 中还有许多内置函数，在以后的学习和使用中你会逐渐接触到。

1.1.4　位运算

位运算的符号是把变量当作二进制数来进行运算的。

在了解位运算之前，我们先来看一看二进制和我们平时使用的十进制有什么相似和不同之处，为熟练使用位运算打好基础。如果已经对二进制很熟悉，可以跳过这一段，直接进入讲解位运算的部分。

我们先以一个十进制数为例。

十进制数 62，它的值由 6*10 + 2*1 组成。我们知道，在十进制数中，个位代表 10 的 0 次方，十位代表 10 的 1 次方，百位代表 10 的 2 次方，千位代表 10 的 3 次方，依此类推。从右到左，第 *n* 个数位上的数字代表着有多少个 10 的 *n*-1 次方相加。对于二进制数来说，这个规律也同样适用，只不过数字从满 10 进一位变成了满 2 进一位。从右到左，一个二进制数的第一位代表 2 的 0 次方，第二位代表 2 的 1 次方，第三位代表 2 的 2 次方，第四位代表 2 的 3 次方……但是，由于一个数位上的数只要满 2 就必须进位，所以二进制数只由 1 和 0 组成。

以十进制数 62 为例（如图 1.3 所示），把它化成一个二进制的数。

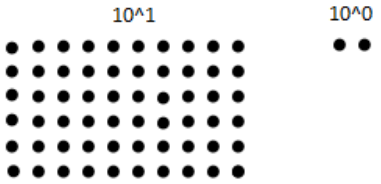


图1.3　用十进制表示62

先把运算过程用图 1.4 表示出来：

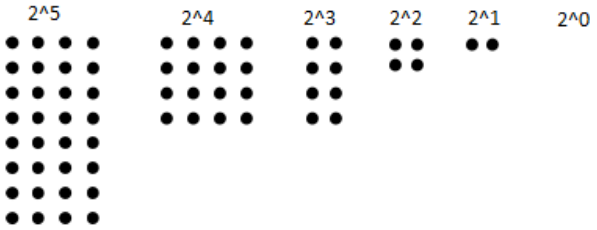


图1.4　用二进制表示62

为了方便理解，把得到的二进制的数每 4 个数位为一组，中间以空格隔开。

62 = 2*2*2*2*2 + 2*2*2*2 + 2*2*2 + 2*2 + 2

62 = 0011 1110

我们看到，62 被拆分成多个 2 的不同次幂的和。每一个幂对应着二进制数的数位上的一个 1，而那些没有出现的幂（比如 2 的 0 次方），它们的数位上则是 0。

所以，二进制数中的 1 和 0 也可以被看作“是”和“否”，就像一个开关一样。像图 1.5 中一样，一个格子就是一盏灯，白色的格子代表灯亮起，亮起的灯就是“是”，代表这个数位上是一个 1，也代表计算这个数的值时要加上这个数位。

图1.5　用灯的亮灭表示二进制数

现在，把十进制数转换为二进制数之后，再试着把它转换回原来的十进制数，验证一下是否正确。

明白了二进制数的概念之后，再来了解位运算的各个符号。

第一个符号是按位与运算符&。参与运算的两个值中，只有对应数位上都是 1 时，那个数位的运算结果才是 1，否则运算结果的对应数位是 0。

用 62 和 21 来做一次与运算，从而更直观地理解按位与运算符。

```
01    a = 62            #a = 0011 1110
02    b = 21            #b = 0001 0101
03    c = a&b           #c = 0001 0100
04    print(c)
```

运行程序，输出结果为：

```
20
```

在 Python 的语法中，我们不能直接定义二进制数，只能定义十进制数，但是进行位运算时，计算机会把十进制数转换为二进制数进行计算，再把二进制的结果转换为十进制数输出。

下面来分析一下这次位运算的过程。

首先，把 a 和 b 这两个数换算成二进制数，再把每一个数位对齐。

```
a :  0011 1110
b :  0001 0101
c :  0001 0100
```

进行按位与运算后，只有（从右到左的）第三位和第五位上，a 和 b 的值都是 1，其他位上要么只有一个 1，要么就都是 0。所以，结果 c 中只有第三位和第五位上是 1，其他位都是 0。把运算结果 c 换算成十进制数，就是 20。

在介绍完位运算具体实现的过程之后，就可以继续了解其他的位运算符号了。

第二个位运算符号是按位或运算符|。它的运算方法和按位与类似，也是对换成二进制的两

个数的对应数位进行运算。不同的是，两个数的数位中只要有一个是 1，结果就是 1。

我们来用运算过程来说明按位或运算符的使用方法：

```
01    a = 62           #a = 0011 1110
02    b = 21           #b = 0001 0101
03    c = a|b          #c = 0011 1111
04    print(c)
```

运行程序，输出结果为：

```
63
```

同样把三个数的数位对齐，来看看按位或运算符的操作过程：

```
a :  0011 1110
b :  0001 0101
c :  0001 1111
```

只有前两个数位上，两个数都没有 1 存在，所以最终答案里只有前两位为 0。最后，把答案 c 换算成十进制数，是 63。

第三个运算符是按位异或运算符^。这次的运算规则是这样的：对应数位上的数不同时，结果中的数位为 1；对应数位上的数相同时，结果中的数位为 0。

直接来看一个应用按位异或运算符的例子：

```
01    a = 62           #a = 0011 1110
02    b = 21           #b = 0001 0101
03    c = a^b          #ans = 0010 1011
04    print(c)
```

同样的，还是把三个数的每个数位对齐来观察按位异或运算的规律：

```
a :  0011 1110
b :  0001 0101
c :  0010 1011
```

从右到左，第一个、第二个、第四个、第六个数位上，要么是 a 的数位的值是 1、b 的数位的值是 0，要么是 a 的数位的值是 0，b 的数位的值是 1。就这样，最终答案就出现了。

最后两个运算符是左移和右移位运算符。左移运算符是<<，右移运算符是>>。这两个运算符的作用，是把一个变量的值化成二进制之后，再把它的整体数位向左移或向右移指定的位数。比如，“a<<2”就是把 a 的二进制数的数位整体往左移动两位之后再转换回十进制数。如果是往左移，那么被覆盖的高位就消失了，而空出来的低位用 0 补上。如果是往右移则刚好相反，被覆盖的低位消失了，空出来的高位则变成了 0。

还是用图来展示这个过程吧，参见图 1.6。

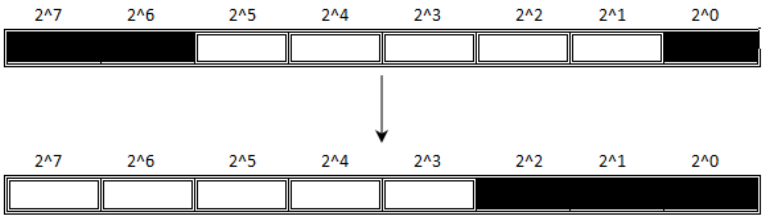


图1.6　二进制数向左移动两位

在图 1.6 中，上面是原来的二进制数，下面是向左移动两位后的二进制数。可以看到，向左移动两位后，原来是 2 的 5 次方到 2 的 1 次方这些数位上是 1，现在变成了 2 的 7 次方到 2 的 3 次方这些数位上是 1。原本 2 的 7 次方和 6 次方数位上的数被覆盖而消失了。

同时，我们发现，原本在 2 的 0 次方数位上的数移动到了 2 的 2 次方的数位上。此时，2 的 1 次方和 2 的 0 次方就空了出来。用 0 填补上这些空位。这样，就完成了一次移动。

看完了图 1.6 之后，再来看一看在程序中该如何实现二进制数的左移和右移。通过下面的例子，你会更直观地了解这两个运算符的作用。

```
01    a = 62            #a = 0011 1110
02    b = 21            #b = 0001 0101
03    print(a<<2)       #a<<2 = 1111 1000
04    print(b>>1)       #b>>1 = 0000 1010
```

运行程序，输出结果为：

```
248
12
```

从程序的注释中可以看到 a、b 以及运算结果转换为二进制数的效果。左移和右移位运算符是可以自行定义不同的移动位数的，所以你也可以尝试设两个数来验证自己对这两个运算符的理解。

1.1.5　使用字符串

可以使用字符串在编程时进行文本处理和判断。本节主要讲解字符串的一些使用和处理方法，涉及一些常用的字符串内建函数。

首先是字符串的拼接。在 Python 中，字符串可以使用“+”进行拼接，就像做数学题的加

法一样。很多时候都需要拼接字符串。比如，如果想把日期和星期几拼接起来，就要用到这个方法。下面展示了解决这个问题的一段程序：

```
01   date = "2018.2.16"
02   day = "Friday"
03   text = date + " " + day
04   print(text)
```

运行程序，输出结果为：

```
2018.2.16 Friday
```

在程序的第 3 行中，新建了一个字符串变量 text，用于把字符串 date 和字符串 day 拼在一起。为了美观，在中间加上一个空格。由此可以直观地得出，字符串的拼接是按从左往右的顺序进行的。

```
01   date = "2018.2.16"
02   day = "Friday"
03   print(date + " " + day)
```

这段程序的输出结果和上一段程序是相同的。

需要注意的是，如果你想在字符串中添加数字，这些数字也必须是字符串类型的变量。

我们来看一个典型的错误例子：

```
01   age = 94
02   text = "Age:"
03   combine = text + age
04   print(combine)
```

其中，age 是一个整型变量，并不是字符串变量，所以运行到第 3 行拼接字符串的时候程序会出错。正确的做法是把 age 转换为字符串类型后再拼接：

```
01   age = 94
02   text = "Age:"
03   combine = text + str(age)
04   print(combine)
```

当然，如果换一种思路，不把 age 转换为字符串也是可以的：

```
01   age = 94
02   text = "Age:"
03   print(text,age)
```

这种方法有一点小缺陷：print 函数的括号内用逗号隔开的变量，在输出时中间会有空格。

在字符串的处理中，大小写的转换也是十分实用的一项技术，这是我们需要了解的第二个方面。

首先来了解 lower 和 upper 这两个字符串内建函数：

```
01    name = "Adam Smith"
02    print(name.lower())
03    print(name.upper())
```

运行这段程序，输出结果为：

```
adam smith
ADAM SMITH
```

观察这段程序可以发现，lower 函数的作用是把字符串中的英文字符全部转换为小写形式，而 upper 函数的作用刚好相反，把所有英文字母都转换为大写形式。

我们仔细地来研究一下这两个函数，函数有时也称作方法。upper 与 lower 相似，我们只以 lower 为例，它的格式为：name.lower()，其中，name 是字符串的变量名，方法 lower()与 name 中间用一个句点连接。方法是 Python 可以对数据进行的操作。句点使 Python 对句点前的变量执行句点后的方法。一般来讲，方法的执行需要额外的信息，在方法名后面的括号内提供。lower 和 upper 方法不需要额外的信息，所以括号内为空，但是必须保留括号。字符串名+句点+(数据)，这是大多数内建函数的格式。

接下来讲解 capitalize 和 title 这两个内建函数。来看下面的程序：

```
01    name = "adam smith"
02    print(name.capitalize())
03    print(name.title())
```

运行程序，输出结果为：

```
Adam smith
Adam Smith
```

根据程序的运行结果可以推断出，capitalize 的功能是把字符串的第一个字母字符变为大写（如果第一个字符不是字母则不变），title 的功能是把字符串内每一个单词的首字母都变为大写，（单词之间由不是字母的字符隔开时，如“good9day”，用 title 方法操作后变为“Good9Day”）。这两个内建函数的点结构和上面所讲的 lower 和 upper 相似。

第三点要讲的是转义字符。为了输出时的美观，常常需要把一串字符分隔开来在不同行输出，或者在不同的字符前添加缩进。如何才能在同一个字符串内实现这些功能而不用多次 print 呢？

请看下面的程序：

```
01    text = "Python\nC\nJava"
02    print(text)
```

运行程序，输出结果为：

```
Python
C
Java
```

在字符串 text 中，\n 叫作换行符，是 Python 中转义字符的一种。这代表 Python 在输出这些字符时，不会输出它们原本的组成字符，而是会输出它们对应的功能，比如\n 对应换行。同样，要想在文本中添加缩进也可以使用类似的方法：

```
01    text = "Programming Language:\n\tPython\n\tC\n\tJava"
02    print(text)
```

运行程序，输出结果为：

```
Programming Language:
    Python
    C
    Java
```

其中，往字符串中添加\t 就是添加横向制表符，从而达到缩进的效果。

除了上面主要讲述的两种转义字符，Python 中还有许多转义字符，如表 1.1 所示。

表 1.1　转义字符表

转义字符	描述	转义字符	描述
\	续行符	\b	退格
\\	反斜杠符号	\e	转义
\'	单引号	\000	空
\"	双引号	\v	纵向制表符
\a	响铃	\r	回车

第四个技巧是如何重复输出字符串。比如，想要重复输出“Python”这个单词三次，每次输出后都换一行。这时可以使用如下方法：

```
01    text = "Python\n"
02    print(text*3)
```

运行程序，输出结果为：

```
Python
Python
Python
```

重复输出字符串的方法就像在做数学中的乘法一样。在合适的时候使用它可以大大提高编写程序的效率。

第五点是编写注释。注释是指存在于程序中但对程序运行效果不产生影响的文字，在程序运行过程中它会被计算机忽略。在编写程序时，如果程序较长或者它的逻辑结构十分复杂，添加注释可以有效地帮助我们看懂程序的结构和作用。妙用注释对于程序员来说有巨大的帮助。

在一行字符前面加上#就可以让这一行出现在#后面的字符全部变成注释。如果需要多行注释，则在每行文字前都加上#即可。

```
01    n = 1+2+3
02    print(n)           #print the value of n
```

需要注意的是，Python 中的注释最好用英文和数字来写。如果在注释中使用中文，则程序在运行过程中有可能出错（即使注释并不是程序要运行的一部分），这取决于编译器的类型和使用的语言。本书中为使大家更好地明白书中要讲解的内容，大部分使用中文编写注释。

1.2　三大结构

Python 语言中有三大结构：循序、分支和循环。这三种结构分别适用于不同的情况，一个复杂的程序中常常同时包含这三种结构。

1.2.1　循序结构

说到“循序”这个词，我们第一个联想到的成语就是“循序渐进”。循序结构和它的名字一样，是按程序内语句的排列顺序运行程序的意思。在前面讲解变量时，我们所举的例子都是循序结构的。这也是 Python 中最简单的结构。

循序结构的运行过程如图 1.7 所示。

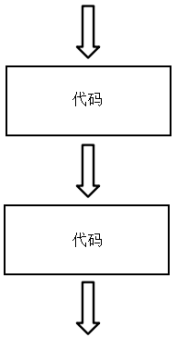


图1.7　循序结构

我们来看一段循序结构的程序吧。如果想让计算机输出我输入的数字的 3 倍，该怎么实现呢？先自己想想，再看下面的程序。

```
01    num = int(input("Enter a number: "))
02    result = num*3
03    print(result)
```

在这段程序中，第一行语句运行完之后，会进入第二行运行，然后再进入第三行。也就是说，排在前面的语句先运行。

1.2.2　分支结构

在 Python 中，分支结构只有一种：if 语句及它的衍生语句。if 语句是根据一条或多条语句的判定结果（True 和 False）来执行对应操作的语句，从而实现“分支”的效果。如上所述，if 语句常常用于程序中的条件判断。

分支结构 if 语句的运行过程如图 1.8 所示。除了最基本的 if 语句，还有 if...else 语句和 if...elif...else 语句，它们可以实现更多的分支和更复杂的逻辑。

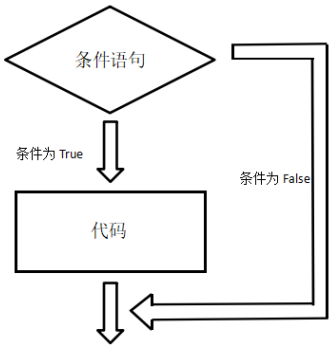


图1.8　if语句

在 if 语句中，如果条件语句的值为 True，那么执行一段程序；如果值为 False，那么不执行这段程序，跳过它直接进入下面的程序。

在 if...else 语句中，如果条件为 True，执行第一段程序，如果为 False，执行第二段程序，然后再进入 if 语句下面的程序，如图 1.9 所示。

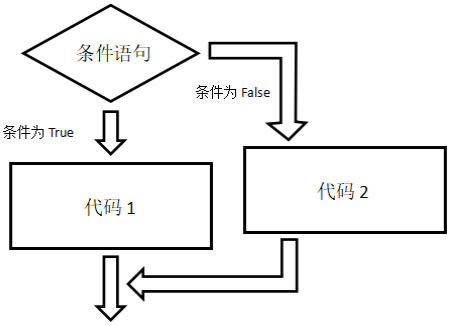


图1.9　if...else语句

在 if...elif...else 语句中，如果第一个条件语句为 True，则执行第一段程序然后直接进入 if 后面的程序；如果第一个条件语句为 False，则继续判断第二个条件语句。如果第二个条件语句为 True，那么执行第二段程序；如果为 False，则执行第三条语句，最后执行完毕，也进入 if 后面的程序，如图 1.10 所示。

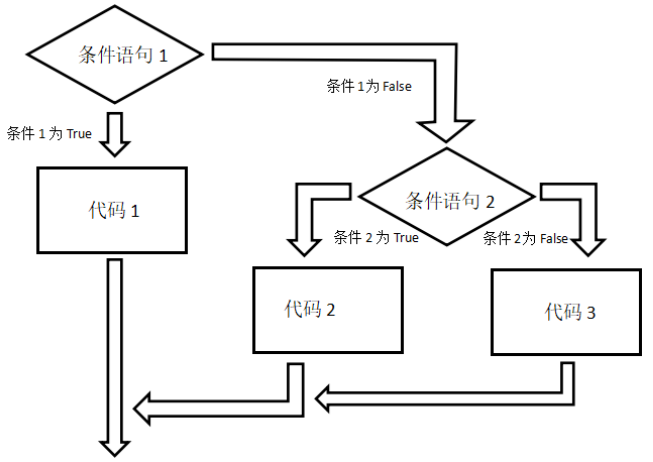


图1.10　if...elif...else语句

在了解分支结构 if 之后，还不能直接开始编写有关分支结构的程序。首先要探究条件语句的返回值，也就是它的判定结果：True 和 False。

1.2.3 条件判断

Python 认定 0 或者 null（空）为 False，任意非 0 值或者任意非空值为 True。这个判定结果也是一种变量——布尔型变量（boolean），它可以被定义或者返回。这种变量的值只有两种：True 和 False。换句话说，这种变量是用来存储对或错的变量。

我们来看一个例子。其中，bool 函数可以把其他类型的变量转换为布尔型变量：

```
01    judge1 = bool(1)
02    judge2 = bool("String")
03    judge3 = bool("")
04    judge4 = bool(0)
05    print(judge1,judge2,judge3,judge4)
```

运行程序，输出结果为：

```
True True False False
```

接下来要了解的是如何在条件语句中检查是否相等，以及如何比较数字的大小。

在 Python 中，判断值是否相等使用“==”，也就是两个等号；判断值是否不相等使用“!=”，也就是一个感叹号和一个等号。

我们用程序来探究这两种语句的返回值：

```
01    num = 5
02    string = "Hello"
03    print(num==5)
04    print(num!=5)
05    print(num==4)
06    print(string!="Hello")
```

运行这段程序，输出结果为：

```
True
False
False
False
```

这段程序中的第一个条件语句 num==5 的返回值是 True，因为 num 的值确实与 5 相等；其他的语句都是不成立的，所以返回 False。不仅仅是数字，其他类型的变量也可以使用是否相等和是否不相等来判断。

现在来比较数字的大小。

先看程序，自己思考一下可能的比较方法，再去看讲解吧！

```
01   year = 2018
02   print(year > 2000)
03   print(year >= 2003)
04   print(year < 2018)
05   print(year <= 2018)
```

运行程序，输出结果为：

```
True
True
False
True
```

显而易见，“>”“>=”“<”“<=”分别对应大于、大于等于、小于、小于等于，和我们熟悉的数学比较一样。如果式子成立，那么返回值为 True；如果不成立，那么返回值为 False。

还可以将多条语句连接起来，从而达到一次性检查多个条件的效果。

先来看一段程序：

```
01   name = "Lucas"
02   judge = False
03   print(name.upper()=="LUCAS" and judge)
04   print(name.upper()=="LUCAS" or judge)
05   print(name == "abc" or True)
```

运行程序，输出结果为：

```
False
True
True
```

在上一节中讲过，upper 函数的作用是把字符串内所有的英文字符都转换为大写形式。所以，name.upper()=="LUCAS"成立，返回的值是 True。然而，由于 and 和 or 两种关键字的功能不一样，最终的结果也不一样。使用关键字 and 时，在 a and b 中，a 和 b 两个值必须都是 True，整条语句的值才能是 True，只要有一个是 False，那么整条语句就是 False。而使用 or 时，在 a or b 中，只要有一个值是 True，整条语句的值就是 True，两个值都是 False 时才会返回 False。所以，因为 judge==False，所以第一条语句的输出是 False，第二条语句的输出是 True。相反，第三条语句的输出中尽管 name 和字符串"abc"并不相等，但因为 or 后面的第二个值是 True，所以输出的值也是 True。

在 Python 中，and 和 or 可以叠加，并默认从左往右运算。就是说，a and b and c or d 这条语句的默认运算顺序为(((a and b) and c) or d)。如果想要更改顺序，比如想先判断 c or d，那么可以加上括号来提高优先级。比如(a and b) and (c or d)就是先判断 a and b 和 c or d，再把这两个结果用 and 进行判断。

1.2.4 应用分支结构

在 1.2.2 节中我们简略地了解了三种分支结构的功能和用法。在本节中，我们将根据几个问题，对 if、if...else、if...elif...else 进行实际运用。

输入出生年份，根据年份判断这个人是否成年（假设今年为 2018 年）：

```
01   year = int(input("Enter the year you born: "))
02   if  2018-year >= 18:              #判断今年（2018）减去出生年份是否大于或等于 18
03      print("You are an adult this year!")
```

我们注意到，if 语句的末尾有冒号，而 print 语句并没有与 if 语句对齐——它的前面有缩进。冒号标志着 if 判断的结束，而 if 下方有缩进的代码是在 True 的条件下要运行的，可以有很多行，一直到缩进再次改变时运行才结束。我们把这一段缩进相同的代码称作代码块。整个程序的缩进方式必须一致，我们通常使用 Tab（按一下）或者空格（按四下）进行缩进。Python 对于缩进和复合语句结尾的冒号要求极其严格，如果弄错会报错。缩进的不同会大大影响一个程序的运行效果。

在这段程序中，print 语句缩进了，所以 print 是 if 内部的代码块。所以，如果 2018-year>=18 为 True，那么计算机会执行 print 语句；反之，如果这条语句的结果是 False，那么程序将会直接跳过 if 结构内的代码，然后结束程序。

我们来看第二个问题：输入一个整数，让计算机判断它是偶数还是奇数，然后输出结果。先想想，用已经学过的语法来写该怎么做呢？

这个问题用 if 语句也可以实现：

```
01   num = int(input("Enter a integer:"))
02   if num%2 == 0:              #如果 num 除以 2 余 0
03      print("Even number")
04   if num%2 == 1:              #如果 num 除以 2 余 1
05      print("Odd number")
```

运行程序，若输入的数字被 2 除的余数为 0 则输出是偶数；若输入的数字被 2 除的余数为 1 则输出是奇数。

上面这段程序中用到了两个 if 语句，但实际上一共只有两种情况，如果余数不为 0 就是 1。为了使程序的运行速度更快，编写起来更方便，可以用 if...else 语句：

```
01   num = int(input("Enter a integer:"))
02   if num%2 == 0:                #如果 num 除以 2 余 0
03      print("Even number")
04   else:                         #if 的判定结果不是 True 时执行 else
05      print("Odd number")
```

这段程序和上一段程序的运行结果是一样的，但只需进行一次 if 判断，如果结果是 True 就运行第一个代码块，如果是 False 就运行 else 里的第二个代码块。在同一个 if...else 语句中，关键词 if 和 else 应当有同样的缩进，而且都要以冒号结束。它们下面的代码块也应该有同样的缩进。

之前我们讲到了有两种分支时该如何编写程序，那么再来看本节最后一个问题：电影院规定，7 岁以下儿童看电影免费，7 岁到 14 岁的儿童半价，60 岁及以上的老人也是半价，其他年龄的观众都是原价。如何才能输入一个人的年龄后判断他买票有什么优惠呢？（年龄小于 0 岁要输出错误提示。）

如果用学过的语法来做的话，要用到 4 个 if 语句。也就是说，就算在第一个 if 语句已经符合条件并打印出结果的情况下，也要继续进行剩余的判断。下面将要用到的 if...elif...else 语句就可以避免这种情况的发生并提高运行速度：

```
01   age = int(input("Enter your age:"))
02   if age < 0:
03      print("Error")
04   elif age >= 0 and age<7:
05      print("Free")
06   elif (age >= 7 and age < 14) or age >= 60:
07      print("Half price")
08   else:
09      print("Full price")
```

elif 就是 else if 的缩写，就是在 else 中再判断一个新的条件。elif 可以无限叠加，直到你想要判断的条件全部判断完。从上往下运行，只要有一个条件符合（True），计算机就会执行这个 if 或 elif 里面的程序，随后跳出整条语句。

只有所有判断语句都不符（False）时，程序才会运行到最后的 else 那里。如果不用考虑全部不符合的情况，else 是可省略的。

1.2.5　循环结构

循环结构是可以多次执行同一段代码的语句结构。在 Python 中有两种循环语句：while 语句和 for 语句。与 Java 和 C 不同，Python 中没有 do...while 循环。

while 语句的结构如图 1.11 所示，在本次循环开始之前先对条件语句进行判断，如果条件语句的值为 True 则循环一次后再次进行判断。如果条件语句的值为 False 则循环结束。这里的条件语句和 if 中的条件语句是一样的。

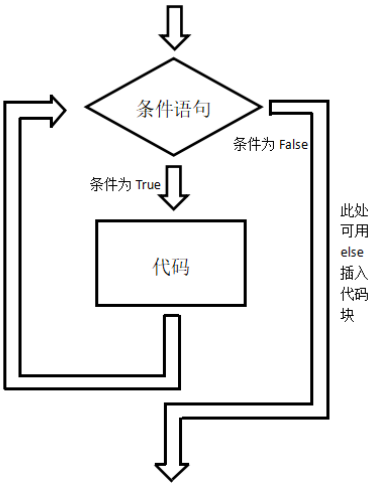


图1.11　while循环

正是由于 while 语句的这种特性，我们常常在不确定循环次数的时候使用它，或者把它写成死循环。此外，还可以用 else 在 while 循环条件为 False 时（此时循环结束）运行另一段代码。

for 语句的结构如图 1.12 所示，从图中看它和 while 语句的结构相似，但其实它们的功能有很多不同。在 for 循环的开始，我们要提供一个范围，无论是一个数的大小范围还是一个列表；循环开始时，会用一个变量存储本次循环的值，每循环一次就更新一次这个变量的值。

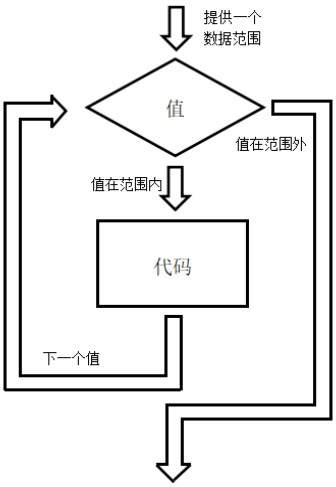


图1.12　for循环

当变量的值不在提供的范围内时，循环结束。for 循环的这种特质，使得它成为我们要确定

循环次数时的首选结构，它用于存储的变量也为我们编写程序提供了很多便利。

与分支结构相似，for 循环也由关键字开头，用冒号结尾，内部的代码块也要有缩进。

1.2.6　continue 和 break

continue 和 break 这两条语句经常出现在循环结构中，它们的作用是让程序跳出这一次循环或是整个循环。由于这种非正常退出循环通常要满足对应的条件，所以 continue 和 break 一般需要配合 if 语句使用。

看一下示例程序来了解如何使用 continue 语句：

```
arr = [1,2,3,4,5,6,7,8,9,10]
for i in range(len(10))
    if arr[i]%5 == 0:
        continue
    print(arr[i])
```

运行程序，输出结果为：

```
1
2
3
4
6
7
8
9
```

我们发现，在列表中能被 5 整除的数都没有被输出。这证明循环运行到能被 5 整除的数时就停下并且退出这一次循环。

把 continue 换成 break 之后，再次运行整个程序：

```
arr = [1,2,3,4,5,6,7,8,9,10]
for i in range(len(10))
    if arr[i]%5 == 0:
        break
    print(arr[i])
```

运行程序，输出结果为：

```
1
2
3
4
```

在列表中出现第一个能被 5 整除的数之后，break 就跳出了整个循环，所以数组后面的几个数就不会被输出了。

1.2.7 应用循环结构

在简单认识了循环结构之后，本节将会把两种循环应用到实践中。

我们要做一个用于计算年龄的程序，让用户输入出生年份和今年的年份，相减得到他今年的年龄。但是，年龄不可能是负数，所以，如果出生年份大于今年的年份，要提醒用户输错了，并允许用户重新输入出生年份和今年的年份。

先自己想想该怎么做，再看正确答案：

```
01   born_year = int(input("Enter the year you born:"))       #输入出生年份
02   this_year = int(input("Enter this year:"))               #输入今年年份
03   while born_year > this_year:                             #当出生年份大于今年年份时循环
04      print("Your age can't be negative. Please enter again.")
05      born_year = int(input("Enter the year you born:"))    #再次输入
06      this_year = int(input("Enter this year:"))
07   else:                                                    #循环结束，数据合法
08      print("Input success.")
09   print("Your age is", this_year - born_year)              #输出结果
```

运行程序，如果输入的出生年份一直大于今年，则计算机仍然执行 while 循环，会让用户一直重新输入，直到输入正确，随后执行 else 语句，然后结束。如果一开始的输入就正确，则不会执行 while 循环，直接进入 else 语句并输出结果。如果不想让 while 循环结束时再执行其他代码，可以删掉 else 语句。

接下来写一段计算一个数的阶乘的程序。数 n 的阶乘写作 $n!$，是这样计算的：$n! = n\times(n-1)\times(n-2)\times(n-3)\times...3\times2\times1$。让用户输入一个数（默认为正整数），然后输出它的阶乘，先自己想想这个问题用 while 循环该怎么实现，再看下面的例子：

```
01   n = int(input("Enter a number:"))      #输入一个数字
02   count = 1                              #定义一个用于计数的变量
03   result = 1                             #定义一个用于存储结果的变量
04   while count <= n:                      #当 count 小于等于 n 时循环
05      result *= count                     #当前的 result 乘上 count
06      count += 1                          #更新 count 的值
07   print(result)
```

在这段程序中，在 while 循环开始前先定义了两个变量：count 和 result。count 用于计算次数，从 1 开始，每次循环都+1，记录本次循环为第几次。result 用于存储计算结果，根据阶乘的

计算方法，我们从 1 开始，一直连续乘到 n，此时 count = n，再加 1 后就大于 n，此时跳出循环并输出结果。需要注意的是，代码中的“result*=count”和“count+=count”是“result = result*count”和“count = count + 1”的简写。

在前面的内容中曾提到，在循环次数确定的情况下还可以使用 for 循环。下面的程序就是用 for 循环实现的，与上一段程序有同样的功能：

```
01   n = int(input("Enter a number:"))          #输入一个数字
02   result = 1                                 #定义一个用于存储结果的变量
03   for i in range(1,n+1):                     #循环时 i 的值从 1 到 n
04      result *= i                             #当前的 result 乘上 i
05   print(result)
```

for 循环的语法格式是“for 迭代变量 in 范围:”，和 while 循环一样有关键词、冒号和代码块的缩进。for 循环默认迭代变量每次循环都+1，变量名称可以自己定义，常用的有 i、j 等，只在这个循环中有效（在循环外无法调用这个变量）。这条语句中的 range 是一个 Python 内置函数，用于定义一个列表，用作数据范围，在 for 循环中极其常用（当然也有 for 循环不需要用到 range 函数的情况，我们会在 1.3 节讲到）。它可以写为 range(x)，range(x,y)或 range(x,y,z)。range(x)的功能是定义一个从 0 到 x−1 的列表作为循环范围，range(x,y)是定义一个从 x 到 y−1 的列表，而 range(x,y,z)则是在定义一个列表的同时，把默认的每次循环变量+1 改成+z（z 可以是负数）。

现在，你知道为什么第 3 行程序中定义的是 range(1,n+1)而不是 range(1,n)了吗？

前面还讲到，while 循环可以用来编写死循环。直接来看程序：

```
01   count = 0
02   while True:
03      print(count)
04      count += 1
```

这个程序永远不会自动结束，因为 while 后面的判断条件永远正确，所以程序会一直输出。

现在，再学习两条语句：continue 和 break。continue 的作用是“跳过这一次循环”，break 的作用是“跳出整个循环”。

为了更加生动地展示它们的功能，我们尝试在上面的程序中加入这两条语句。

```
01   count = 0
02   while True:
03      if count%5 == 0:                #如果 count 是 5 的倍数
04         count += 1                   #count 的值+1
05         continue                     #跳过这次循环进入下一次
06      elif count == 12:               #如果 count 的值为 12
07         break                        #跳出这个循环
```

```
08        print(count)
09        count += 1
```

运行程序，输出结果为：

```
0
1
2
3
4
6
7
8
9
11
```

我们发现，程序并没有无限循环下去，而是在输出 11 后结束了；并且，5 的倍数都没有被输出。这是因为，我们用 if 语句在 for 循环中设置了分支，如果 count 为 5 的倍数，则 count+=1 后 continue，也就是回到 while 语句，直接进入下一次循环，不再执行代码块后面的程序。

而当 count 的值为 12 时，用 break 跳出这个循环，到达程序结尾，也没有执行程序块后面的程序。

1.2.8 结构的嵌套

在 1.2.7 节中，我们看到了循环结构 while 和分支结构 if 的嵌套。本节会讲解另外两个结构互相嵌套的例子。

写一个用于做整数除法的程序，让用户输入被除数和除数。由于除数不能为 0，所以，如果输入的除数是 0，输出错误提示并结束程序。而除法的结果可能是整数或小数，在输出结果的同时，也输出结果的类型。

先自己思考一下该怎么做，再来看下面的程序：

```
01    num1 = int(input("Enter the dividend:"))
02    num2 = int(input("Enter the divider:"))
03    if num2 != 0:                                  #如果输入的除数不是 0
04        result = num1/num2
05        if result == int(num1/num2):               #如果 result 的值与两数相除取整后的值相等
06            print(int(result),"integer")
07        else:
08            print(result,"decimal")
09    else:
10        print("The divider can't be 0.Error. ")
```

这一段程序中用到了两个 if...else 语句，其中一条语句在另一条语句的内部，也就是说，第一个 if 语句的代码块中包含了另一条 if 语句。这样的写法同样适用于其他结构。

我们再来看第二个例子。在一所学校内，学生都有自己单独的学号，它们由两个两位数组成。但是，这些学号有一个规律：假设学号为 ab cd，其中两位数 ab 不超过 20，两位数 cd 不超过 30，同时两位数的开头不能是 0。要把所有的学号都输出该怎么做？

同样，先自己思考，再看下面的程序：

```
01    for i in range(10,21):
02        for j in range(10,31):
03            print(i,j)
```

用两个 for 循环嵌套起来，程序很快就写完了。这意味着，第一个 for 循环每循环一次，都要把第二个 for 循环全部走完。这比起把所有学号都自己枚举出来然后一起输出要快得多。

1.3　列表

列表是 Python 中五种标准变量类型的一种。一系列按顺序排列的元素组成了列表，一个列表中可以包含字符、数字、字符串，甚至子列表。

在 Python 中，同一个列表中可以包含多个不同类型的元素，也就是说，不同类型的变量可以存在于同一个列表内，如图 1.13 所示。列表的长度以及列表内元素的值都是动态的——它们可以随时被修改。

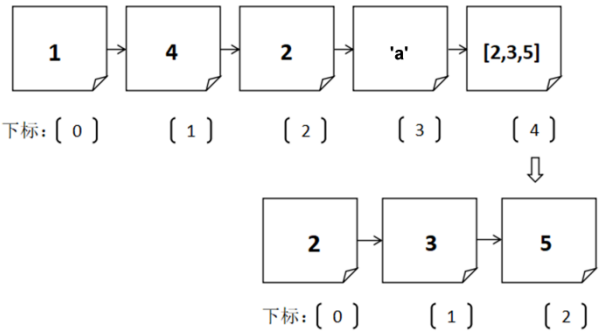


图1.13　包含列表的列表（list4）

1.3.1　定义列表

在 Python 中，列表用中括号表示（[ ]）。

下面的程序中定义了 4 个不同的列表：

```
01    list1 = [ ]
02    list2 = [2,3,5,7]
03    list3 = ["a","b","c"]
04    list4 = [1,4,2,"a",[2,3,5]]
```

很明显，list1 是一个空列表，它的内部不包含任何元素。通常在需要提前声明变量类型为列表的情况下，用定义空列表来实现。第 2 行的 list2 是一个整数列表，它的每一个元素都是一个整型变量，而 list3 中的每一个元素都是一个字符串。最后，list4 是一个包含多种元素的列表，可以看到，它内部有数字、字符串和列表这三种元素。

1.3.2 对元素进行操作

现在已有一个包含多种元素的列表：

```
01    mylist = [2,3,5,7,11,13,17,19,"integer","string","list",[123,456]]
```

列表内包含多种元素，那么如何调用这些列表内的元素进行计算和拼接等工作呢？

由于列表是一个有序集合，所以可以根据索引来找到每一个元素。元素的格式是列表的名称+[索引]，索引从 0 开始。比如，列表 mylist 的第一个元素就是 mylist[0]。因为第一个元素的索引为 0，所以最后一个元素的索引应该是列表的长度减 1，如图 1.14 所示。

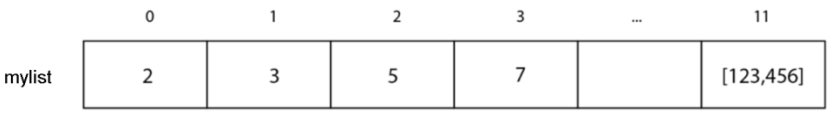


图1.14　列表mylist

如果列表中的元素还是一个列表，那么如何调用子列表内的元素呢？很简单。比如在列表 mylist 中，最后一个元素是 mylist[11]，这个元素是一个列表。那么，这个子列表的第一个元素就是 mylist[11][0]。

下面是对列表 mylist 内的元素进行的一些操作，并对结果进行输出：

```
01    mylist = [2,3,5,7,11,13,17,19,"integer","string","list",[123,456]]
02    print(mylist[11][0]/mylist[1])
03    print(mylist[8]+" "+mylist[9])
```

运行程序，输出结果为：

```
41.0
integer string
```

其中，mylist[11][0]是指 mylist 中第 12 个元素（子列表）的第 1 个元素，也就是 123；mylist[1]是指 mylist 的第 2 个元素，也就是 3。它们相除的商就是 41.0。同理，mylist[8]和 mylist[9]是指列表中的第 9 个元素和第 10 个元素，它们和一个空格拼接起来，就成了输出结果的第 2 行“integer string”。

也可以修改列表中元素的值：

```
01   mylist = [2,3,5,7,11]
02   mylist[4] = 9
03   print(mylist,mylist[4])
```

运行程序，输出结果为：

```
[2,3,5,7,9] 9
```

列表内的元素和变量一样，值可以被随时更新，也可以被用来赋值给其他变量。

列表中的元素个数是非固定、可以改变的。也就是说，可以往列表中添加新的元素，也可以把元素从列表中移除。先来看看如何往列表中添加元素。

给定一个元素为“2,3,5,7,13,17,19”的列表 mylist，这个列表内的所有元素都是质数。要想把这个列表转换为从 2 到 23 的有序质数列表，就要用到 append 和 insert 这两个内置列表函数。

如图 1.15 和图 1.16 所示，内置函数 append 的功能是在列表的末尾插入新的元素，而函数 insert 的功能是在指定索引的位置插入新的元素，原本在这个索引位置处的元素及其后面的所有元素往后顺移一位。

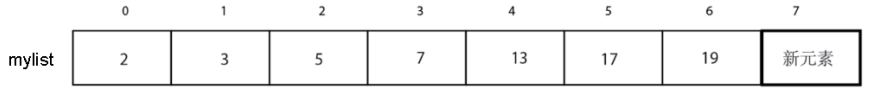


图1.15　append函数

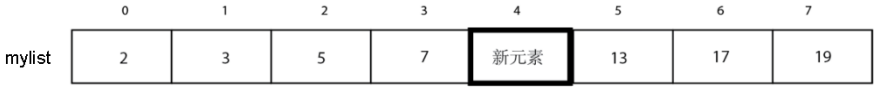


图1.16　insert函数

在下面的程序中，每往列表中添加一个元素，就把整个列表打印出来，方便观察规律：

```
01   mylist = [2,3,5,7,13,17,19]
02   mylist.append(23)
03   print(mylist)
04   mylist.insert(4,11)
05   print(mylist)
```

运行程序，输出结果为：

```
[2,3,5,7,13,17,19,23]
[2,3,5,7,11,13,17,19,23]
```

可以看到，在 append 函数执行完毕后，列表的长度增加了一位，23 被加在了列表的最后。而 insert 函数执行完后，新元素 11 被插到索引为 4 的位置，也就是 mylist[4]的位置——第 5 个元素。原本排在第 5 个的元素 13 并没有消失，它和它后方的元素全部往后移动了一位，也就是索引增加了 1。由这个例子也可以看出，列表内置函数的调用方法是：列表名+句点+函数名+小括号，括号内是函数执行所需要的数据。

append 函数的括号中填写需要插入在列表末尾的元素，insert 函数括号中的第一个数据填写要插入的索引值，第二个数填写需要插入的元素，中间用逗号隔开。这些语法都是固定的，只有按照正确的方法编写程序才能按照设想运行。

再来看看如何从列表中移除元素。

如果要移除列表中的元素，有两个函数可以选择：remove 和 pop。remove 函数的标准格式为：list.remove(obj)，它的功能是移除指定列表（list）中某个值（obj）的第一个匹配项。pop 函数的标准格式为：list.pop(index)，它的功能是移除指定列表（list）中某个索引位置（index）上的元素，同时也返回被移除的元素的值。这两种函数运行完毕——指定元素被移除后，它后方的所有元素向前移动一位，也就是索引减少 1，如图 1.17 所示。

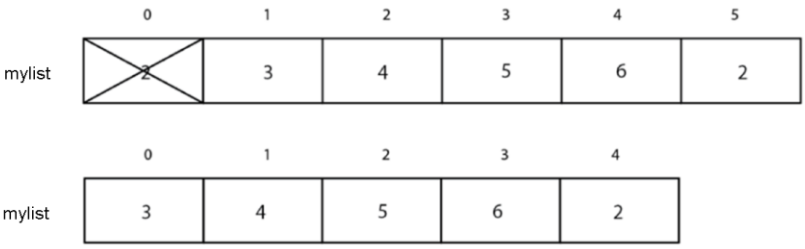


图1.17　移除元素

下面是一个具体的例子：

```
01   mylist = [2,3,4,5,6,2]
02   mylist.remove(2)
03   print(mylist)
04   num = mylist.pop(0)
05   print(mylist,num)
```

运行程序，输出结果为：

```
[3,4,5,6,2]
[4,5,6,2] 3
```

在这段程序中，remove 函数移除了列表 mylist 中匹配到的第一个等于 2 的元素。由于列表的遍历顺序是按索引大小从小到大进行的，被移除的是第一个 2，而不是最后的 2。第 4 行中的 pop 函数移除了当前列表索引为 0 的数，也就是第一个数 3，同时把 3 返回。

编写程序时我选择用变量 num“接住”传输回来的元素，并将它和 pop 操作后的列表一起输出，这就叫弹出元素。弹出功能可以在我们需要一次性使用列表中的数时使用（用完一次就丢掉）。

也可以选择不用变量保存 pop 的返回值，而把它单纯地作为一个用于移除元素的函数使用。如果 pop 的括号内没有填写要弹出的元素的索引（空括号），则默认弹出列表的最后一个元素。

要移除列表中的元素还有一个方法：

```
01   numbers = [1,2,3,4,5,6]
02   del numbers[2]
03   print(numbers)
```

运行程序，输出结果为：

```
[1,2,4,5,6]
```

del 函数的使用格式为“del+空格+要删除的元素”，它和 pop 函数的删除方法相似，需要提供列表名和索引，但它没有 pop 函数返回被删除的元素值的功能。

1.3.3　列表的顺序

我们常常需要把字符串或者数字按首字母顺序和数字大小排序。如何把大小随机的元素按顺序排列呢？如图 1.18 所示。

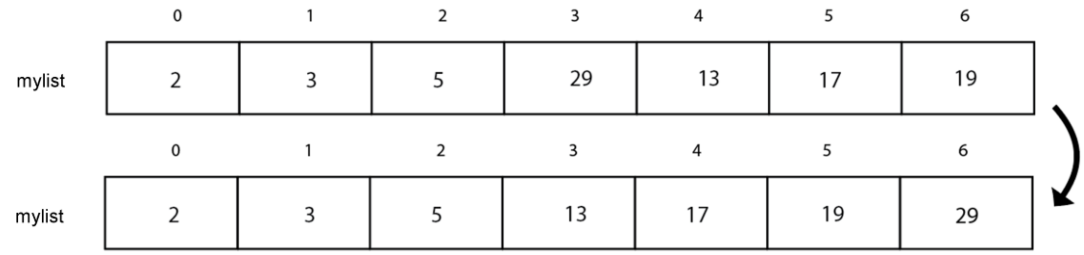


图1.18　排序

只需要一步：

```
01   mylist = [2,3,5,29,13,17,19]
02   mylist.sort()
03   print(mylist)
```

运行程序，输出结果为：

```
[2,3,5,13,17,19,29]
```

经过列表内置函数 sort 的操作后，mylist 被进行了永久性的从小到大的排序——原来的顺序无法复原了。如果列表里的元素都是字符串，则 Python 会自动按首字母从 a 到 z 的顺序排列。需要注意的是，如果用 sort 函数进行排序，那么列表内的元素必须都是同一种类型，否则程序会出错。

那么，如果想要从大到小排序呢？

```
01   mylist = [2,3,5,29,13,17,19]
02   mylist.sort()
03   mylist.reverse()
04   print(mylist)
```

运行上面的程序，输出结果为：

```
[29,19,17,13,5,3,2]
```

在这段程序中，首先用 sort 函数对 mylist 进行了永久性的从小到大排序，然后用内置函数 reverse 把整个列表的元素的顺序全部反过来。也就是说，对 mylist 使用了 reverse 函数后，最后一个元素变成了第一个元素，倒数第二个元素变成了第二个元素，依此类推。和 sort 函数相同，reverse 函数的反转操作也是永久性的。在 sort 函数的括号中可以填入其他函数并按照函数内部规定的方法排序，如图 1.19 所示。

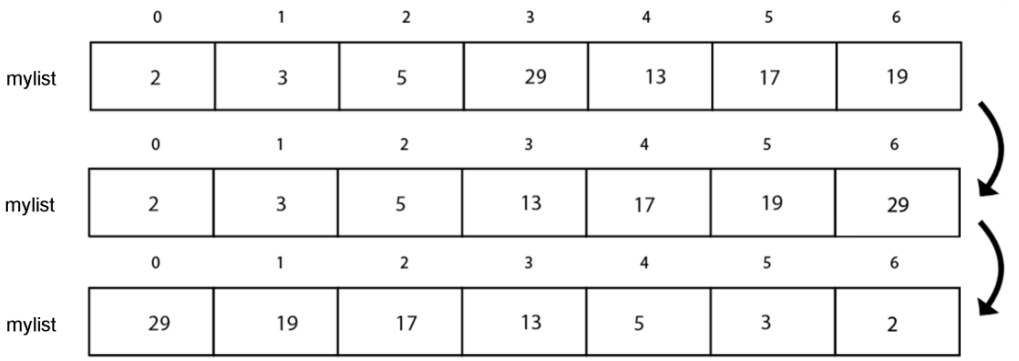


图1.19　从大到小排序

1.3.4　列表内置函数

除了前面几节讲到的 append、insert、remove 等内置函数，列表还有很多其他非常实用的内置函数。

首先要讲的是 len 函数。函数的名字“len”其实是英文单词 length 的缩写。length 是长度的意思，顾名思义，这个函数是用来求列表的长度的。下面是一段使用 len 函数的例子：

```
01    mylist = [2,3,5,29,13,17,19]
02    L = len(mylist)
03    print(L)
```

运行程序，输出结果为：

```
7
```

函数 len 的使用格式是 len(列表名)，它会返回列表的长度，也就是列表里元素的个数。在上面的程序中，列表的长度是 7。由于 len 函数有返回值，所以如果你想多次使用它，最好用一个变量存储它的值。如果只是想要输出，可以忽略第 2 行，直接调用 print(len(mylist))就可以了。

接下来要了解的是 max 函数和 min 函数。这两个函数可以找出列表中的最大值和最小值：

```
01    mylist = [2,3,5,29,13,17,19]
02    print(max(mylist),min(mylist))
```

运行以上程序，输出结果为：

```
29 2
```

同样，如果想要多次使用 max 函数或 min 函数返回的值进行计算，也要用一个新的变量存储它们的值。

第 4 个要介绍的函数是 count，它是用来统计指定元素在指定列表中的出现次数的函数：

```
01    mylist = [2,3,5,29,13,17,19,2,2,3]
02    print(mylist.count(2),mylist.count(3),mylist.count(19))
```

运行以上程序，输出结果为：

```
3 2 1
```

count 函数的使用格式为：列表名+句点+函数名（count）+括号（括号内为需要统计出现次数的元素的值）。可以看到，在列表 mylist 中，值为 2 的元素有 3 个，值为 3 的元素有 2 个，值为 19 的元素有 1 个，所以 count 函数的返回值分别为 3、2、1。如果指定元素在列表中不存在，那么 count 函数的返回值为 0。

第 5 个函数是 index 函数，它可以在列表中找出与指定值匹配的第 1 个元素的索引位置。

```
01   mylist = [6,2,8,7,2,3,4]
02   print(mylist.index(2))
```

运行以上程序，输出结果为：

```
1
```

根据前面讲到的几个内置函数的规律，index 函数也有返回值，使用格式也和 count 函数相似，括号内是需要查找的元素的值。在列表 mylist 中，第一个值为 2 的元素是第 2 个元素，也就是说，索引是 1，所以 index 函数返回了 1。即使索引为 4 的元素的值也是 2，但是由于它并不是第一个匹配的元素，所以 index 不会返回它的值。

第 6 个函数是 clear 函数，它可以把列表清空：

```
01   numbers = [1,2,3,4,5,6]
02   numbers.clear()
03   print(numbers)
```

运行以上程序，输出结果为：

```
[]
```

clear 函数的使用格式为：列表名+句点+clear()。clear 函数不需要额外的数据。

最后要讲到的函数是 copy 函数，copy 函数的功能是复制原有的列表。

说到复制列表，大家心里应该已经有了一些不用这个函数也可以进行复制的思路：

```
01   numbers = [1,2,3,4,5,6]
02   copylist = numbers
03   print(numbers,copylist)
```

这段程序的输出结果为：

```
[1,2,3,4,5,6] [1,2,3,4,5,6]
```

输出结果一样，是不是就复制成功了呢？

我们来看看对复制完的列表中的元素进行改动会如何：

```
01   numbers = [1,2,3,4,5,6]
02   copylist = numbers
03   copylist.pop()
04   print(numbers,copylist)
```

输出结果为：

```
[1,2,3,4,5] [1,2,3,4,5]
```

你会发现，只移除了 copylist 列表的最后一个元素，可是 numbers 列表的最后一个元素也被移除了。如果复制成功，两个列表应该是两个互不影响的个体，这又是为什么呢？

这是因为 Python 中列表存储的方法和其他语言中的不太一样，列表中的元素在计算机的存储空间中占据一定的内存，而列表本身存储的是这些元素的存储地址，在调用列表元素的时候根据地址来调出它们原本的值。如果直接给新的列表赋值，只是复制了原来列表存储的地址，所以元素本身并没有被复制成两份，如图 1.20 所示。

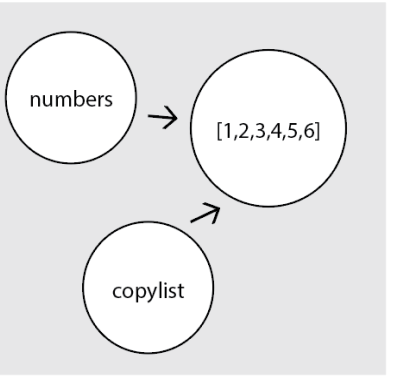


图1.20　复制列表

要复制元素本身就需要使用内置函数 copy：

```
01   numbers = [1,2,3,4,5,6]
02   copylist = numbers.copy()
03   copylist.pop()
04   print(numbers,copylist)
```

运行这段程序，输出结果为：

```
[1,2,3,4,5,6] [1,2,3,4,5]
```

copy 函数返回的是复制原列表中的元素后产生的一组新元素的存储地址，其被存储在新的列表中。这样，修改复制后的列表中的元素就不会影响到原来的列表了，如图 1.21 所示。

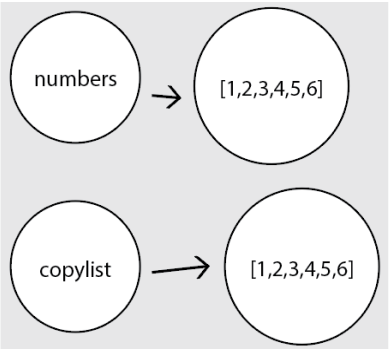


图1.21　copy函数的效果

1.3.5　截取和拼接列表

当只需要使用列表的一部分或想把两个列表合并成一个时，可以对列表进行截取和拼接操作。下面是对列表进行截取操作的演示程序：

```
01    numlist = [2,3,4,6,1,3,5,7,9,10,22,18,31,24,53]
02    print(numlist[:5],numlist[2:5],numlist[5:])
```

运行程序，输出结果为：

```
[2,4,3,6,1] [4,6,1] [3,5,7,9,10,22,18,31,24,53]
```

截取操作过程如图 1.22 至图 1.24 所示。

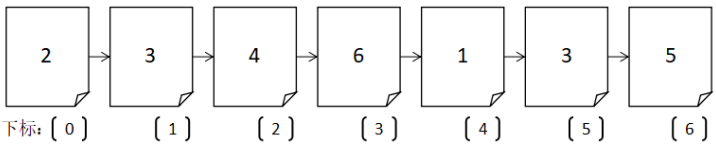


图1.22　完整的列表numlist

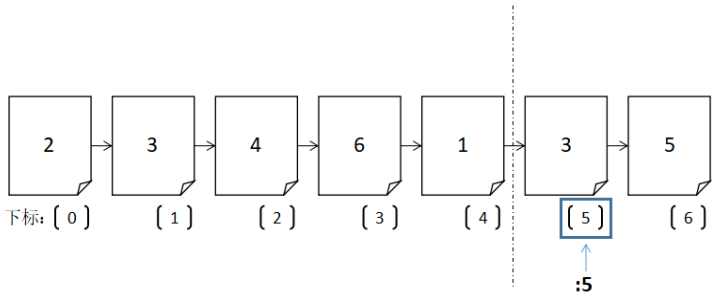


图1.23　numlist[:5]

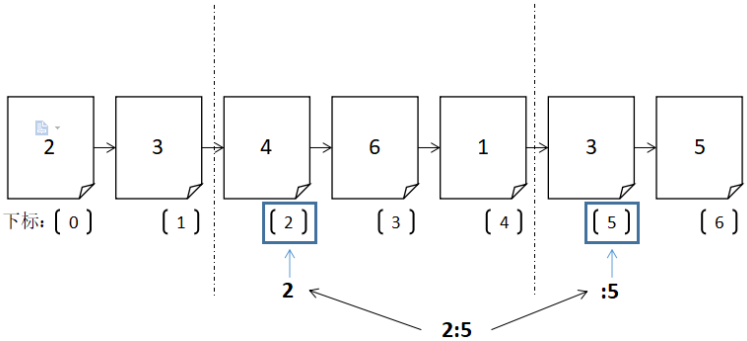


图1.24　numlist[2:5]

截取列表的方式为：列表名+[*m*:*n*]，其中 *m* 必须小于 *n*，否则截取到的列表为空。截取到的元素范围为列表中索引值从 *m* 到 *n*−1 的元素。如果没有填写 *m*，则 *m* 默认为 0，也就是从列表的开头截起，如果 *n* 没有填写则 *n* 默认为列表的长度（len(列表名)），也就是截取到列表的结尾。

列表也可以被拼接：

```
01    numlist = [2,3,4,6,1,3,5]
02    newlist = numlist[:5] + numlist[2:5]
03    print(newlist)
```

运行程序，输出结果为：

```
[2,3,4,6,1,4,6,1]
```

拼接操作如图 1.25 所示。

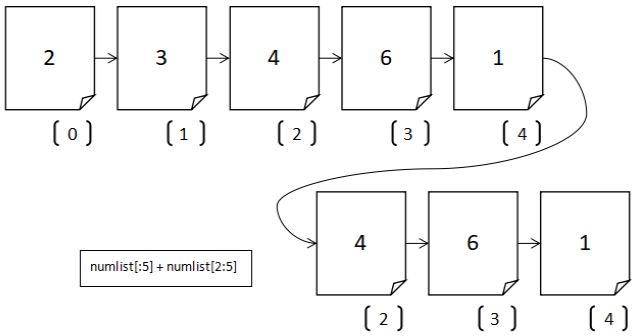


图1.25　截取的列表片段相加

新的列表是旧的两个列表拼接起来得到的，这两个列表可以用截取的方式截出，也可以是

一个列表变量或者函数返回的列表。和直接赋值不同，对拼接得到的列表进行改动不会影响原来的两个列表。

1.3.6 字符串、元组和列表

这三种变量类型其实有很多相似之处。在讲解这些相似处之前，我们先来了解元组这种变量类型。

元组和刚刚学习的列表十分相似，它们都是 Python 五种标准变量类型中的一种，也同样是序列类型的变量，即由有序排列的空、单个或多个元素组成的变量，它们内部的元素都可以被调用，元素的种类可以不同。它们的不同之处在于，元组中的元素是固定不变的，它们的值不能被更改。元组的长度也是固定的，即不能添加或删除元素。

我们使用小括号来定义元组。元组和列表一样，可以进行拼接和截取：

```
01   numtuple = (2,4,6,8,10)
02   print(numtuple)
03   newtuple = numtuple*2
04   print(newtuple)
05   print(newtuple[2:5])
```

运行程序，输出结果为：

```
(2,4,6,8,10)
(2,4,6,8,10,2,4,6,8,10)
(6,8,10)
```

程序中的第三行，numtuple*2 相当于 numtuple + numtuple。

元组可以被重复赋值，但是其中的元素不可以被更改：

```
01   numtuple = (2,4,6,8,10)
02   print(numtuple)
03   numtuple = (2,4,6,8)
04   print(numtuple)
05   print(numtuple[0])
06   #非法操作：
07   #numtuple[0] = 100
08   #numtuple.append(10)
```

运行程序，输出结果为：

```
(2,4,6,8,10)
(2,4,6,8)
2
```

由此可见，元组的合法操作和列表完全相同。以上程序用注释写出了两个对元组非法操作的例子。如果这两条语句不是注释，则程序运行的时候会出错。所以，要想修改元组内部元素，只能给元组重新赋值或者在定义元组的那一行直接修改程序。

我们熟悉的字符串其实也与元组十分相似。可以把字符串描述成由一个或多个字符组成的元组，它不能被修改，但可以被截取、拼接或者赋值。但是，字符串只能由字符构成，而元组可以由多种类型的变量构成，如图 1.26 所示。

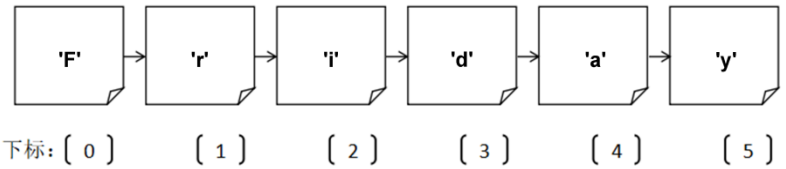


图1.26　字符串的结构

下面的程序演示了字符串的截取、拼接功能的使用：

```
01    str1 = "abcd"
02    print(str1[0],str1[1])
03    print(str[1:3])
04    print(str1[1:3]+str1[:2])
```

运行程序，输出结果为：

```
a b
bc
Bcab
```

字符串、元组和列表是可以相互转换的。这时候要用到 str、list 和 tuple 函数，这三个函数可以返回转换后的字符串、列表和元组。我们来看一个字符串转换为元组和列表的例子：

```
01    sample = "abcdefg"
02    print(list(sample),tuple(sample))
```

运行程序，输出结果为：

```
['a','b','c','d','e','f','g']
('a','b','c','d','e','f','g')
```

可以看到，字符串中的每一个字符都被转换成单个元素存储在列表和元组中。任何字符串都可以转换为列表和元组。

再来看看列表转元组、元组转列表以及列表和元组转字符串的例子：

```
01    list1 = ["abcdefg","hijk"]
```

```
02   tuple1 = (1,2,3,4,5)
03   print(tuple(list1),list(tuple1))
04   print(str(list1),list(tuple1))
05   del list1[0]
06   tuple = (1)
07   print(str(list1),str(tuple))
```

运行程序，输出结果为：

```
('abcdefg','hijk') [1,2,3,4,5]
['abcdefg','hijk'] (1,2,3,4,5)
hijk
1
```

可以看出，列表和元组相互转换没有任何问题，而元组和列表转字符串时会有不同的情况出现。当列表或元组内有多个元素时，它们转成的字符串中会包括定义时输入的所有字符，包括中括号、小括号、逗号和用于表示字符串的引号。当列表或元组内只有一个元素时，这个元素会转成正常的字符串，不包含中括号或小括号或者用于标记字符串的引号。

1.3.7 用循环遍历列表

如果想要把列表中的值依次拿出来使用或者依次输入，可以用 for 循环遍历列表。

先来看一道题目：写一段程序，让用户把整数输入到列表中，再把列表内的整数全部相乘。

首先，编写一段程序让用户输入数字，并把它们加入列表：

```
01   numbers = []                                          #定义一个空列表
02   total = int(input("Enter the amount of numbers:"))    #输入列表内要有多少个整数
03   for i in range(total):                                #用 for 循环把元素加入列表
04      numbers.append(int(input("Enter a number:")))
05   print(numbers)                                        #输出列表检查
```

运行程序，自己输入一些数据，对照输出结果检查程序是否正确。

然后，继续编写程序，把整个列表中的数相乘。

```
01   numbers = []                                          #定义一个空列表
02   result = 1                                            #定义一个变量用于存储结果
03   total = int(input("Enter the amount of numbers:"))    #输入列表内要有多少个整数
04      for i in range(total):                             #用 for 循环把元素加入列表
05      numbers.append(int(input("Enter a number:")))
06   for i in range(total):                        #用 for 循环把列表中的每个元素与 result 相乘
```

```
07        result *= numbers[i]                                    #i 为每个元素的下标
08    print(result)
```

运行程序，输入数据，检查输出结果是否符合预期。

由于任何数乘 0 都等于 0，所以 result 的初始值只能为 1。同时，由于列表内元素的下标从 0 开始到长度减 1，所以可以直接用 range(total)来设定循环范围。

这道题还有另一种做法：

```
01    numbers = []                                          #定义一个空列表
02    result = 1                                            #定义一个变量用于存储结果
03    total = int(input("Enter the amount of numbers:"))    #输入列表内要有多少个整数
04    for i in range(total):                                #用 for 循环把元素加入列表
05        numbers.append(int(input("Enter a number:")))
06    for i in numbers:                                     #用 for 循环把列表中的每个元素与 result 相乘
07        result *= i                                       #i 为列表的每个元素本身
08    print(result)
```

这段程序的运行结果与上一段相同，但让列表内数据相乘的方法却不同。i 循环的范围是列表 numbers，那么 i 每一次循环时的值就是列表中的某个元素：第 1 次循环时 i==numbers[0]，第 2 次循环时 i==numbers[1]等。这样，每次循环直接把 result 乘上 i 就可以了。

1.3.8　字典简介

字典也是 Python 五种基础变量类型中的一种，它和列表一样，可以同时存储多个任意类型的元素，所以放在列表这里进行讲解。但是，比起相同之处，它和列表的不同之处更多。现实中的字典，比如英汉词典，是用来根据英语单词来找它对应的中文释义的，而 Python 中的字典是根据输入的关键字在内部寻找对应的值并返回，如图 1.27 和图 1.28 所示。

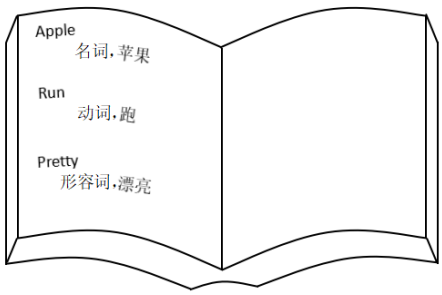


图1.27　字典

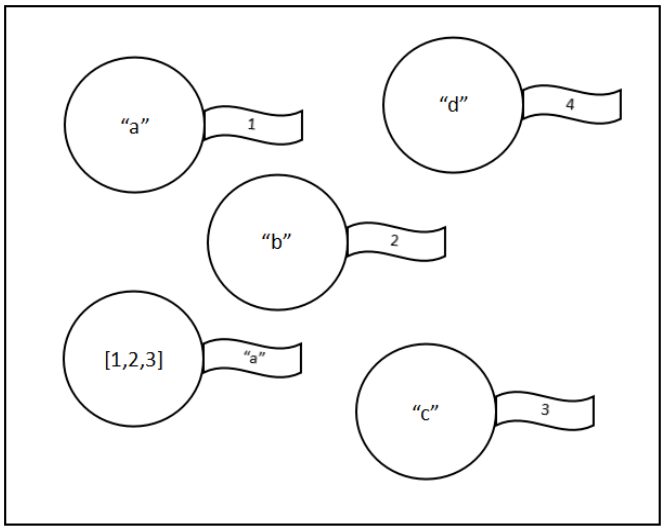


图1.28　字典中元素的调用和顺序及大小无关

就像列表使用中括号，元组使用小括号一样，定义字典或给变量赋值字典时使用大括号{}。

通过下面这一段程序来了解元组的定义和使用。

```
01    letters = {1:"a",2:"b",4:"d",3:"c","a":123}          #给字典定义一个初始值
02    print(letters)                                       #打印整个字典
03    print(letters[2], letters[4], letters["a"])          #调用字典元素
```

运行这段程序，输出结果为：

```
{1:'a',2:'b',4:'d',3:'c','a':123}
b d 123
```

定义字典时，每个元素叫作一个键值对，用 x:y 的形式定义，其中 x 为每个元素的关键字，也就是键；y 为元素储存的值。键在字典中必须是唯一且不可变的，而不同键代表的值可以是任意数据类型，也可以有相同的值存在。

使用字典时，和列表不一样，在中括号的内部输入的不是元素的索引值，而是元素的键。比如，键 2 所对应的值是字符串"b"，那么输出 letters[2]就是输出"b"的意思；同理，键 4 和键"a"也是一样的。如果中括号内的键在字典中不存在，那么程序会报错。想要更改键所对应的值，方法为：比如 letter[4] = "m"，这样键 4 所对应的值就从之前的"d"变为了"m"。

也可以在字典中添加新的元素，但方法和我们之前了解的有点不同。添加新元素的办法是添加新的键，不需要使用 append 函数：

```
01    letters = {1:"a",2:"b",4:"d",3:"c"}     #给字典定义一个初始值
```

```
letters[5] = "e"                          #给字典加入新的元素（键值对）
letters["abc"] = 6
print(letters)                            #输出字典
```

运行程序，输出结果为：

```
{1:'a',2:'b',4:'d',3:'c',5:'e','abc':6}
```

用程序中的这种格式编写代码，新的键值对就会自己出现在字典中了。

可以对列表使用的函数 len、del、clear 和 copy 等，对字典也可以使用，方法是相同的。它还有更多的功能等待你去了解。

1.4　函数

函数是指用于实现一个特定功能并能重复使用的代码段。前面学到的 input 和 print 这些函数是 Python 语言的内建函数，可以直接使用。也可以自己定义函数来实现我们想要的功能，这样的函数叫作用户自定义函数。本节就来了解用户自定义函数。

1.4.1　定义子函数

要想使用函数，必须先定义它。

定义函数的格式为：

def 函数名(变量名):

其中，变量名指传进子函数的数据在子函数内部对应的变量名，可以没有，也可以有一个或者多个。函数名是在调用函数时要使用的名字，两个函数不能有相同的变量名。也就是说，在 Python 中已经存在 print 函数，所以在创建自定义函数时不能以 print 作为变量名，但是可以使用 Print 作为变量名（一个字符大写了）。

最后，如果这是一个有返回值的函数，要在函数的最后写上“return 返回值”。

长篇大论不如一个例子，这就来创建一个函数看看。

```
def Plus(a,b):
    print(a+b)

def Hello():
    num1 = int(input("Hello! Please enter the first number:"))
    num2 = int(input("Please enter the second number:"))
    return num1,num2
```

```
08
09    n1,n2 = Hello()
10    Plus(n1,n2)
```

这是一个用于计算两个数的和的程序。第一个函数 Plus 用于计算两个数的和，第二个函数 Hello 用于向用户问好并输入两个数字。

Plus 函数被调用时（第 10 行），需要提供两个值。a 对应 n1 的值，b 对应 n2 的值，在函数内部使用这两个值时只能调用 a 和 b，不能调用 n1 和 n2。如果调用语句是 Plus(3,5)，那么 a==3、b==5，a 和 b 继承它们对应位置上的值。这个函数没有返回值，只需要 print，所以在函数结尾不需要写 return。

从 Hello 函数的定义语句中可以看到，括号内并没有变量名，这代表调用函数时不需要提供任何值。在函数内，num1 和 num2 存储了两个输入的值，用 return 返回。Python 和其他语言不同，可以同时返回多个值，而且可以是不同类型的值。返回多个值时，在等号的左边用多个变量接住，变量之间用逗号隔开；同样，变量继承它们对应位置上的值。在程序中，n1 继承了 num1 的值，n2 继承了 num2 的值。

注意：由于程序是从上到下运行（行数编号从小到大）的，所以函数必须在调用语句的上面定义。也就是说，程序第 9 行的代码如果被换到第 3 行，程序就会报错，因为这时 Hello 函数还没有被定义。另外，和 if、while 和 for 语句相似，函数的定义语句下的代码段也要有缩进。

在这段程序中，做加法的简单任务不创建函数其实更简单。在复杂的程序中，使用自定义函数可以使解决问题的过程简便直观得多。

主函数的定义是程序的入口函数。其他的函数（不是主函数的函数叫作子函数）和方法都在主函数中调用。

1.4.2 主函数

在 C 语言中，主函数和子函数之外的语句只能是定义语句。而在 Python 中，函数之外也可以有定义语句之外的语句，比如输出等。Python 程序运行时只会运行不在函数内的代码和主函数内的代码段，不会运行子函数内的代码段。子函数内的代码段只有在被调用时才会运行。

因为有这个特性，如果程序的调用和被调用范围仅限于本文件内，可以不编写主函数，直接在所有子函数下面编写代码就可以。但是，为了养成好的习惯和直观性，建议在编写程序时加上主函数。本书为了简单易懂，有些代码段不使用主函数。

下面我们来了解一下在 Python 中如何定义主函数。

```
01   if __name__ == '__main__':
02      print("Inside")
```

运行程序，输出结果为：

```
Inside
```

if __name__ == '__main__'是定义主函数的语句。定义函数的时候要注意代码块前面的缩进（本质上是 if 语句的格式）。

再来看一个例子：

```
01   def sq(num):
02      print(num*num)
03   print("Outside")
04   if __name__ == '__main__':
05      print("Inside")
```

运行程序，输出结果为：

```
Outside
Inside
```

在函数外和主函数内的输出语句都被执行了，而子函数 sq 因为没有被调用，其中的输出语句并没有被执行。

1.4.3　调用函数

在 1.4.1 节介绍子函数时，就在程序中调用了我们定义的子函数。但是，不同的子函数有不同的调用方法。本节就来讲解调用各种函数的规律。

在定义子函数时，还要在函数的括号中定义好需要传进子函数的变量。调用函数的格式和为函数定义的需要传进函数的变量有关。调用函数时传进的数据是什么类型的变量，函数内部对应位置上的变量就会是同样类型的。

比如，我们定义了一个函数 calc，需要三个变量，最后返回的值是第一个变量和第二个变量的乘积减去第三个数的值，如图 1.29 所示。

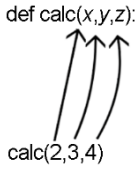


图1.29　函数calc

函数定义和调用时的格式是这样的：

```
01   def calc(a,b,c):
02      ans = a*b - c
03      return ans
04   ans = calc(2,3,4)
05   print(ans)
```

运行程序，输出结果：

```
2
```

如果在调用函数时写的是 calc('2',3,4)，那么函数就会在执行的时候出错，因为'2'对应的是 a，3 对应的是 b，4 对应的是 c，string 类型的变量无法和 int 类型的变量进行计算。

值得注意的是，在调用函数时，我们还使用了一个变量来存储函数的返回值。需不需要这个变量取决于函数的内容。如果函数有返回值，也就是有 return 语句的话，就需要用一个变量存储这个返回值；反之，如果没有返回值，那么就可以直接调用，不需要变量来存储返回值。

如果想要输出这个返回的值，或者再使用它进行其他一次性操作，也可以不专门用一个变量存储，而直接把这个函数写入进行操作的语句中。再拿 calc 函数举例说明：

```
01   def calc(a,b,c):
02      ans = a*b - c
03      return ans
04   print(calc(2,3,4))
```

运行程序，输出结果为：

```
2
```

这样，就达到了简化程序的目的。

再来举一个没有返回值的函数的例子：

```
01   def Output(strArr):
02      for i in range(10):
03         print(strArr[i])
04   Arr = ['Apple','Bear','Cat','Dog','Egg','Fruit','Giraffe','Hat','Ice','Jacket']
05   Output(Arr)
```

运行程序，输出结果为：

```
Apple
Bear
Cat
Dog
```

```
Egg
Fruit
Giraffe
Hat
Ice
Jacket
```

1.4.4　全局变量

在之前见到的程序中，子函数内部的变量和子函数外部的变量完全不相关，所以不必担心重名的问题；子函数外的语句无法调用子函数内部的变量，子函数内部的语句也无法调用子函数外部的变量。这使程序的结构更加清晰，但也带来了一些不便：难道想使用外部的数据就必须要传入子函数中吗？

全局变量，顾名思义，就是可以让整个程序使用的变量。它一般在整个程序的最开始定义，在所有的函数和语句的上方。如图 1.30 所示，局部变量只在函数的小范围内存在，跳出这个范围，局部变量不存在。而全局变量在整个程序的范围内存在。

图1.30　全局变量与局部变量

再次用 calc 函数举例说明：

```
01   g = 10
02   def calc(a,b,c):
03       ans = a*b - c
04       return ans
05   print(calc(2,3,4))
```

在这段程序中，g 就是一个全局变量。在主函数和自己定义的子函数之外的所有语句中，都可以像调用一个平常变量一般调用它。但是，在子函数内部时，就要给它多加一条语句：

```
01   global g
```

global 的意思是“全球的”，在程序中也可以被理解为“全局的”。在子函数内，在变量名前面加上 global 这条语句用于声明这是一个全局变量。有了这条语句，这个子函数内的所有代码都可以像各个子函数外的代码一样调用这个变量并对它进行更改，而且更改的结果也会在全局范围内保存。与调用函数时传进函数的变量不同，在函数内更改这些非全局变量并不会影响它们本来的值。

1.4.5 函数的运用

我们来看一个问题：1 个苹果 2 元，1 个菠萝 8 元，1 个西瓜 10 元。1 个盒子 1 元，它最多能装下 5 个苹果或 1 个菠萝或 1 个西瓜。编写一个程序，让用户输入每种水果的购买数量，然后输出每种水果加上盒子需要多少钱，以及一共要多少钱。

先自己思考一下该怎么做，再来看下面的代码。

首先是没有使用函数的解决方法：

```
numA = int(input("Enter the number of apples:"))    #输入
numP = int(input("Enter the number of pineapples:"))
numM = int(input("Enter the number of watermelons:"))

if numA//5 < numA:                      #a//b 指 a 除以 b 的商的整数部分
    box = numA//5 + 1                   #box 存储需要盒子的个数，box 不能是小数，所以要+1
else:
    bag = numA//5
sumA = 2*numA + box                     #sumA 用于存储苹果总价
print("Price of apples:",sumA)          #输出苹果总价
sumP = 8*numP + numP                    #sumP 用于存储菠萝总价
print("Price of pineapples:",sumP)      #输出菠萝总价
sumM = 8*numM + numM                    #sumM 用于存储西瓜总价
print("Price of pineapples:",sumM)      #输出西瓜总价
print("Total:",sumA+sumP+sumM)
```

然后是使用了函数的解决方法：

```
def Apple(n):                       #函数 Apple 内 n 为购买的苹果个数
    if n//5 < n/5:                  #a//b 指 a 除以 b 的商的整数部分
        box = n//5 + 1              #box 存储需要盒子的个数，box 不能是小数，所以要+1
    else:
        box = n//5
    print("Price of apples:",2*n+bag)       #输出苹果总价
    return 2*n+box

```

```
09   def Pineapple(n):                    #函数 Pineapple 内 n 为购买的菠萝个数
10      print("Price of pineapples:",8*n+n)      #输出菠萝总价
11      return 8*n+n
12
13   def Watermelon(n):                   #函数 Watermelon 内 n 为购买的菠萝个数
14      print("Price of watermelons:",10*n+n)   #输出西瓜总价
15   return 10*n+n
16
17   numA = int(input("Enter the number of apples:"))
18   numP = int(input("Enter the number of pineapples:"))
19   numM = int(input("Enter the number of watermelons:"))
20   print("Total:",Apple(numA)+Pineapple(numP)+Watermelon(numM)) #三个函数返回值的和
```

现在，你熟悉子函数的定义和使用方法了吗？

根据图 1.31 所示的关系来写一个满足问题要求的程序。

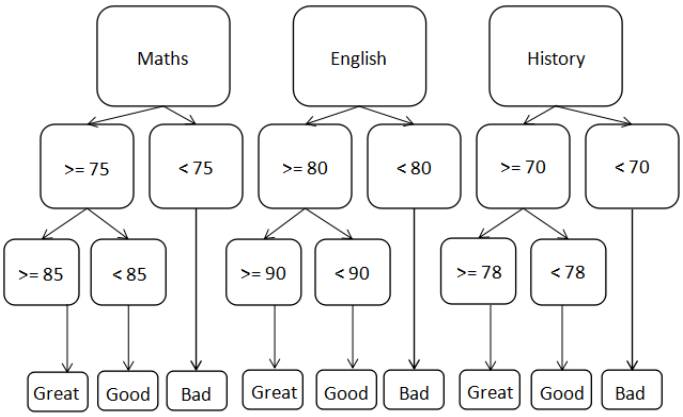


图1.31　各科分数评价

首先，允许用户输入数学、英语、历史这三科的考试分数，每一科的分数范围在 0~100 之间，超出这个范围的分数是错误分数，需要重新输入。第二，对每一科的分数做出评价，评价标准如图 1.31 所示。

第三，按表 1.2 中规定的比例折算各科分数并得到总分，总分范围是 0~300。

表 1.2　各科折算总分的比例

	折算比例
History	80%
English	110%

续表

	折算比例
Maths	110%
Total	300%

第四，按照表 1.3 中的标准，给总分一个等级。最后，输出对每一门课考试成绩的评价、折算完的总分数和总分对应的等级。分别用 5 个子函数实现这 5 个要求。

表 1.3　总分对应等级

	>=270	>=240	>=210	>=190	<190
总分对应等级	A+	A	B+	B	C

这是一个比较复杂的题目，提供的数据和条件比较多。先自己思考该怎么做，用纸和笔记录下自己的思路和计划，再动手去写代码。写完代码后，再对照书上的答案验证。记住，不同的代码也可以实现相似的功能，书上的答案仅供参考。

那么，首先来写出这个程序的框架：

```
01   def JudgeSubjects(mscore, escore, hscore):
02      next
03
04   def JudgeTotal(total):
05      next
06
07   def CalculateTotal(mscore, escore, hscore):
08      next
09
10   def InputScore():
11      next
12
13   def OutputResult(total, TotalLevel, SubjectLevel):
14      next
15
16   if   name   == "  main  ":
17      math, english, history = 0,0,0
```

先定义好所有要使用的子函数和主函数。语句 next 是用来“跳过”一个子函数的。当子函数内部没有任何语句时，Python 就会把下一个子函数或者主函数当作子函数内部的语句，从而因为缩进不对而报错，加上 next 语句就可以有效地防止这种情况的发生。同时，根据子函数的功能来判断它们需要什么值来完成任务，从而确定括号内的变量。

现在，就把这些子函数填满。

```
def JudgeSubjects(mscore, escore, hscore):
    if mscore >= 75:
        if mscore >= 85:
            mlevel = 'Great'
        else:
            mlevel = 'Good'
    else:
        mlevel = 'Bad'

    if escore >= 80:
        if escore >= 90:
            elevel = 'Great'
        else:
            elevel = 'Good'
    else:
        elevel = 'Bad'

    if hscore >= 70:
        if hscore >= 78:
            hlevel = 'Great'
        else:
            hlevel = 'Good'
    else:
        hlevel = 'Bad'

return {'Maths':mlevel,'English':elevel,'History':hlevel}

def JudgeTotal(total):
    if total >= 270:
        result = 'A+'
    elif total >= 240:
        result = 'A'
    elif total >= 210:
        result = 'B+'
    elif total >= 190:
       result = 'B'
    else:
        result = 'C'
    return result

def CalculateTotal(mscore,escore,hscore):
    return mscore*110/100 + escore*110/100 + hscore*80/100
```

```
43
44   def InputScore():
45      math = int(input("Enter your Maths test score:"))
46      while math < 0 or math > 100:
47        print("The score is invalid.")
48         math = int(input("Enter your Maths test score:"))
49
50      english = int(input("Enter your English test score:"))
51            while english < 0 or english > 100:
52               print("The score is invalid.")
53            english = int(input("Enter your English test score:"))
54
55      history = int(input("Enter your History test score:"))
56            while history < 0 or history > 100:
57               print("The score is invalid.")
58      history = int(input("Enter your History test score:"))
59
60      return math,english,history
61
62   def OutputResult(total,TotalLevel,SL):
63      print('Total Score:',total)
64      print('Total Level:',TotalLevel)
65      print('MathsLevel:',SL['Maths'],'EnglishLevel:',SL['English'],'HistoryLevel:',SL['History'])
66
67   if __name__ == "__main__":
68      math,english,history = InputScore()
69      SubjectLevel = JudgeSubjects(math,english,history)
70      total = CalculateTotal(math,english,history)
71      result = JudgeTotal(total)
72      OutputResult(total,result,SubjectLevel)
```

到这里，整个程序就完成了。

看完这两个问题之后可以发现，如果没有注释的话，没有使用函数的解决方法虽然也不难完成，但是在写完一大段连着的代码之后很容易混淆代码的用处。而使用了函数就可以有效避免这种情况的发生。所以，使用函数还可以让程序变得更有条理。

到这里，第 1 章就结束了。在接下来要学习的诸多算法中，将会大量使用本章讲解的语法。在进入下一章之前，一定要先熟悉这些语法，可以多做题目来练习。如果你觉得还没有弄懂，那就请重新读一读本章再继续接下来的学习吧！

第 2 章

双指针问题

“指针”是编程语言中的一个对象，它存储着一个内存空间的地址，计算机可以通过这个地址找到变量的值。也就是说，这个特定的地址指向这个特定的值。指针最大的优点在于它可以有效利用零碎的内存空间。本章主要涉及的问题有如下几项。

- 数组合并：合并两个有序数组。
- 二分查找：在有序数组中查找元素。
- 链表：链表的概念和应用。

2.1 数组合并

在第 1 章中，我们了解了列表的使用方法。通过列表，可以建立内部只存在整型变量的数组。可以用数组来模拟指针。

2.1.1 合并有序数组

指针的意思是内存空间的地址。可以通过一个数组中每个元素的下标来找出它的值，所以存储这个元素位置的下标值的变量可以看作一个指针。我们将以这个概念来实现 Python 中的指针问题。由于它不是真正意义上的指针，所以我们叫它“模拟指针问题”。

我们先来看第一个问题：数组合并问题。

有两个从小到大有序排列的数组，如图 2.1 所示。

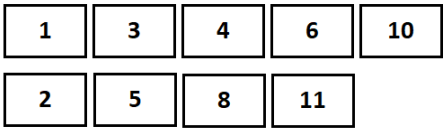


图2.1　两个有序数组

第一个数组里有 5 个元素，第二个数组里有 4 个元素。要想把它们合并成一个新的从小到大排列的数组，该怎么编程实现呢？

在开始编写程序之前，我们先来了解一下这道题的思路。

首先拿出第二个数组的第一个元素 2，把它和第一个数组中的第一个元素进行比较，如图 2.2 所示。

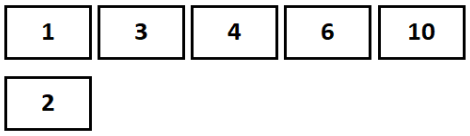


图2.2　第一次比较

2 比 1 要大，所以移动到第一个数组的第二个元素 3，再次进行比较，如图 2.3 所示。

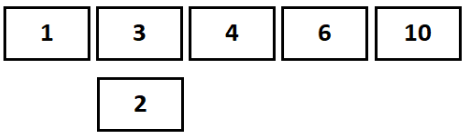


图2.3　第二次比较

2 比 3 要小，而我们已经知道 2 比前面的 1 要大，所以可以把 2 插到 3 的位置。3 和它后面的元素都向后移动一个位置，如图 2.4 所示。

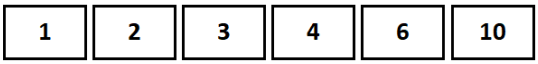


图2.4　向第一个数组中插入2

这时第二个数组中还剩下三个元素 5、8、11。把 5 拿出来从 3 开始比较。注意，因为两个数组都是从小到大有序排列的，2 在原来的第二个数组中排在 5 前面，所以它和它前面的数字都比 5 小，直接从 3 开始比较即可，如图 2.5 所示。

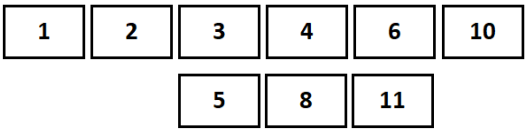


图2.5　第二个元素

5 在 6 这里停下，因为 6 大于 5。把 5 插到 6 的前面，如图 2.6 所示。

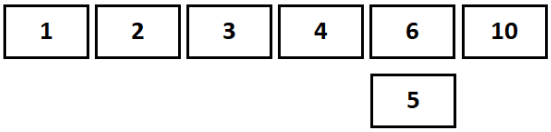


图2.6　准备插入第二个元素

完成插入后，第一个数组如图 2.7 所示。

图2.7　把5插入第一个数组

随后，再重复之前的操作，直到第二个数组中的所有元素都被插入第一个数组中。此时排序完成，结果如图 2.8 所示。

图2.8　排序完成

现在来把这个过程转换为程序。

首先需要一个 for 循环对第二个数组 arr2 实现从小到大的遍历：

```
for i in range(0,len(arr2)):
```

随后在 for 循环内部加上 while 对 arr1 进行操作：

```
01    ind = 0
02    for i in range(0,len(arr2)):
03        while ind < len(arr1):                #ind 的范围不能超过数组元素下标的最大值
04            if arr2[i] <= arr1[ind]:
05                ans.insert(ind + i,arr2[i])  #向第一个数组中插入第二个数组中的数
06                break
07            else:
08                ind += 1                      #如果 ind 指向的数比 i 指向的数小，则 ind 向后一位
```

注意：Python 中的数组也叫列表，为了方便理解统一称为数组。

其中，ind 是存储数组中数值的下标的变量，ans 用于存储排序完毕后的数组中的变量（初始值为 arr1）。为什么不使用 for 循环而要用 while 呢？因为同一个数前面可能要插入两个数字，而 for 循环只能让它的前面插入一个数。

2.1.2 最终代码

用指针合并两个有序数组的程序如代码 2.1 所示。

代码 2.1 用指针合并两个有序数组

```
arr1 = [1,3,4,6,10]                      #初始化两个数组
arr2 = [2,5,8,11]
ind = 0
ans = arr1.copy()                        #ans 初始化为 arr1
for i in range(0,len(arr2)):
    while ind < len(arr1):
        if arr2[i] <= arr1[ind]:         #ind 的范围不能超过数组元素下标的最大值
            ans.insert(ind + i,arr2[i])  #向第一个数组中的合适位置插入第二个数组中的数
            break
        else:
            ind += 1             #如果 ind 指向的数比 i 指向的数小，则 ind 向后一位
    else:                        #如果 arr1 已遍历完，直接把剩下的 arr2 拼到 arr1 结尾
        ans = ans + arr2[i:]
        break
```

其中，while...else 语句是用来判断 while 循环是否被完整执行完的语句。如果 while 循环的结束是因为 while 后面的判断语句（ind < len(arr1)）的返回值为 False，则执行 else；如果是因为 break 而跳出循环，则不执行 else。

有人可能会有疑问：为什么要用 ans 存储 arr1 的值，而不在 arr1 内部直接改动呢？如果直接使用 arr1 存储答案，在向数组中添加元素的过程中，列表内部的元素会变化，也就是说，我们丢失了 arr1 的原来的值。用 ind 调用原来列表中的元素与 arr2 中的元素进行比较，而向 ans 中插入 arr2 的数，就可以有效避免这个问题。

这样，就使用模拟指针完成了两个有序数组的合并。

2.2 二分查找

用模拟指针的方法完成了数组的合并后，我们再来看看如何用指针实现有序数组的二分查找。

2.2.1　什么是二分查找

第二个问题也是有关有序数组的问题。如何快速在一个有序数组中准确地找到一个元素的位置呢？这就要用到本节要讲到的中心知识点——二分查找了。

二分查找又叫作折半查找，意思为每次查找后，查找的范围都折半。这样查找到最后，查找范围内只剩一个数时，判断它是否为要查找的数。如果是，就记录它的位置；如果不是，则要查找的数不在这个数组中。

现在，用图来了解一下二分查找。有一列体积从小到大有序排列的球。我们想要找到其中体积为 4 的球，该怎么做呢？

如果使用普通的遍历方法，如图 2.9 和图 2.10 所示，需要对每一个球进行判断，直到找到体积相同的球为止。用这种方法会使耗时大大增加。在元素较少的情况下，消耗的时间差异并不明显，但当整个序列中元素较多时（如上万个元素），使用二分查找的优势就十分明显了。

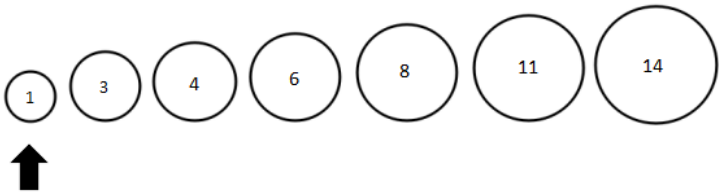


图2.9　普通遍历查找

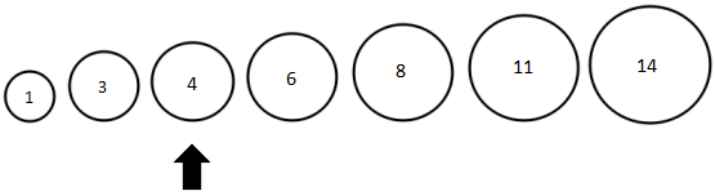


图2.10　普通遍历查找（开始到结束，从左到右）

我们来看看换成二分查找的做法，把要查找的球从体积为 4 的球换成体积为 6 的球。

如图 2.11 所示，二分查找需要两个指针，一个指向数组的第一个元素，叫作头指针；另一个指向数组最后一个元素的后方，叫作尾指针。假如最后一个元素的下标是 7，那么尾指针存储的值则是 8。要查找的范围是在头指针和尾指针之间的所有元素（包括头指针指向的元素但不包括尾指针指向的元素）。

初始化之后，需要找到查找范围中的中间数，也就是处于中间位置的元素。可以通过把头指针和尾指针的值相加再除以 2 来得到中间数的下标值。在这个问题中，(0+8)/2 = 4。把下标 4

指向的元素与要查找的元素做比较：8 > 6。

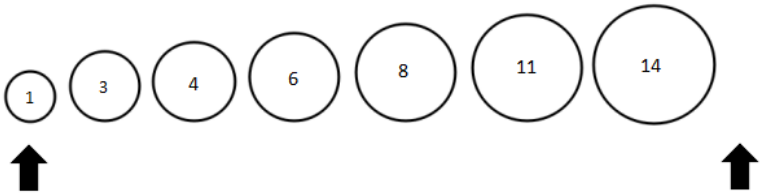


图2.11　二分查找初始

因为这是一个从小到大有序排列的数组，所以要查找的元素必定在 8 前面，它的下标一定小于 4。所以，给尾指针赋值 4，让它指向 8，把查找范围缩小到下标 0~4，进入图 2.12 所示的状态。

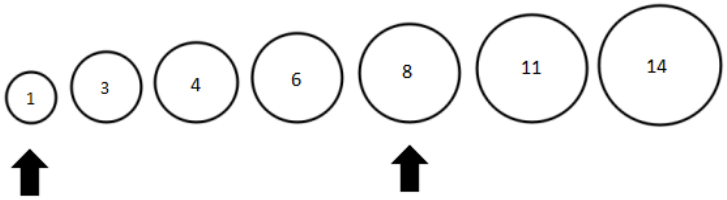


图2.12　二分查找—移动尾指针

再一次对头指针和尾指针存储的下标值进行计算，求出中间数的下标，(0+5)//2 = 2（只取整数部分）。下标 2 指向的元素为 4，把 4 和 6 做比较，4 < 6。与之前同理，要查找的元素必定在 4 后面，它的下标一定大于 2。所以，我们给头指针赋值 2 来缩小查找范围，如图 2.13 所示。

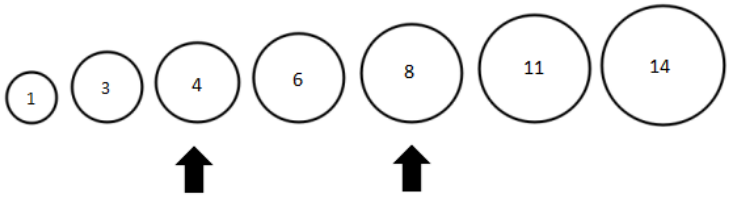


图2.13　二分查找—移动头指针

重复之前的操作，计算得出中间数的下标为 3。把下标 3 指向的数与 6 对比，发现 6 = 6。这时，就可以结束查找并记录元素的下标 3 了。

2.2.2　问题求解

在 2.2.1 节中，我们用一个长为 7 的数组演示了二分查找。在本节中，我们要用程序实现一

个有 15 个元素的数组[1,3,5,6,7,8,13,14,15,17,18,24,30,43,56]的二分查找。

由于整个查找过程将要重复相似的操作，并且不确定总共的查找次数，因此使用 while 循环来进行查找。当头指针和尾指针之间只剩一个数时，不需要再重复进行查找而可以直接进行判断，所以把 while 循环的条件设为下面的语句：

```
while tail - head != 1:
```

然后要编写头指针和尾指针之间只剩一个数时用于判断的语句。这里也用到了本章第一小节中提到的 while...else 语句：如果 while 循环因为 break 而结束，证明查找已经完成，不需要再判断一遍了。

```
01   while tail - head > 1:    #当尾指针 tail 减头指针 head 等于 1 时，查找范围内只有 head 指向的数
02      next                   #由于 while 循环中还没有编写代码，用 next 暂时代替
03   else:
04      if search == numbers[head]:    #search 是要搜索的元素
05         ans = head
06      else:
07         ans = -1            #如果数组中没有这个元素，那么输出-1
```

有人可能会问，为什么不用判断尾指针指向的数，而只用判断头指针指向的数呢？在题目的开始可以看到，尾指针的定义是指向查找范围中最后一个数的后面的指针，它本身是不包含在查找范围内的。

接下来要编写 while 内部的语句。每一次 while 都是一次操作，最后的结果要么是移动头指针，要么是移动尾指针，要么是直接找到结果。这段话应该已经给了你提示，要采用 if...elif... 语句来实现。下面就是我们的程序了：

```
01   while tail - head > 1:    #当尾指针 tail 减头指针 head 等于 1 时，查找范围内只有 head 指向的数
02      mid = (head+tail)//2  #mid 存储中间数的下标，//2 代表对/2 的结果舍去分数部分取整
03      if search < numbers[mid]:      #search 是要搜索的元素，如果它小于 mid 指向的元素
04         tail = mid
05      elif search > numbers[mid]:    #如果 search 大于 mid 指向的元素
06         head = mid + 1              #mid 指向的元素小于 search，所以不用把它保留在范围内
07      elif search == numbers[mid]:
08         ans = mid
09         break                       #找到元素的话就直接结束
10   else:
11      if search == numbers[head]:
12         ans = head
13      else:
14         ans = -1                    #如果数组中没有这个元素，那么输出-1
```

2.2.3 最终代码

二分查找的最终代码如代码 2.2 所示。

代码 2.2 用指针实现有序数组中的二分查找

```
numbers = [1,3,5,6,7,8,13,14,15,17,18,24,30,43,56]
head,tail = 0, len(numbers)        #数组长度刚好是最大下标值+1
search = int(input("Enter a number to search:"))

while end - start > 1:    #当尾指针 tail 减头指针 head 等于 1 时，查找范围内只有 head 指向的数
   mid = (head+tail)//2 #mid 存储中间数的下标，//2 代表对/2 的结果舍去分数部分取整
   if search < numbers[mid]:       #search 是要搜索的元素，如果它小于 mid 指向的元素
      tail = mid
   if search > numbers[mid]:       #如果 search 大于 mid 指向的元素
      start = mid + 1              #mid 指向的元素小于 search，所以不用把它保留在范围内
   if search == numbers[mid]:
      ans = mid
      break                        #找到元素的话就直接结束
else:
   if search == numbers[head]:
      ans = head
   else:
      ans = -1                     #如果数组中没有这个元素，那么输出-1
print(ans)
```

你可以随意更改数组中已有的元素，也可以随意输入数值来验证二分查找。二分查找问题到这里就解决了。

2.3 链表

链表是用指针连接的用于存储数据的数组，它最大的优点在于可以有效利用零碎的内存空间。在很多语言中，数组的大小要提前定义，定义后不能随便更改，而且数组中只能存储同一类型的变量。

如果使用了链表，则可以改变数组的长度，并且可以在同一个数组中存储不同类型的元素。说到这里，你一定想起了 Python 语言中的列表吧。实际上，列表的工作原理就是链表。由于 Python 语言中没有指针，所以将要在本节中用模拟指针的方法来实现链表。

2.3.1 什么是单链表

链表的每个元素不仅仅存储这个元素的值，还要存储与它相连的元素的指针的值，这样链

表才能被连起来。单链表的每个元素包含一个本身的值和一个指向下一个数的指针。因为链表的最后一个数没有下一个数，所以它的指针为空指针。

图 2.14 所示的是一个单链表的示例，这是这个链表通过指针连接顺序排列后的结果。

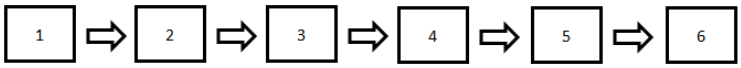


图2.14　单链表示例

由于使用的是模拟链表，所以在原本的数组中，数字的顺序可能像图 2.15 这样，是一个无规律排列的数组。

图2.15　数组原排列

但是，使用指针之后，就变成了图 2.16 这样。

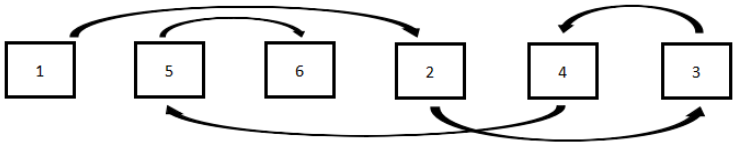


图2.16　用数组中的元素排成单链表

从没有指针指向的第一个元素开始，跟着箭头走到没有存储下一个元素的指针的最后一个数，就完成了单链表的遍历。

2.3.2　建立单链表

接下来，我们建立一个单链表，并按链表的顺序把所有元素输出一遍。

因为我们的链表是模拟链表，所以有不止一种建立链表的方法。首先来看第一种方法，把链表中的每一个元素存储的值存储在一个列表里，再把它们的指针存储在另一个列表里。

```
01    ListValue = [1,5,6,2,4,3]
02    ListPointer = [3,2,-1,5,1,4]
```

在两个数组中，相同下标的元素组成了链表中的一个元素。也就是说，链表的第一个元素的值是 1，它在 ListValue 这个存储数值的数组中的下标是 0，则 ListPointer[0] == 3，它是链表中这个元素存储的指针。

这个指针指向下一个元素，也就是 ListValue 中下标为 3 的数值 2。和 2 属于链表中同一个

元素的指针是 ListPointer[3]，它指向下一个数值。依此类推，直到有一个指针的值为-1，代表它的后面没有其他元素了。ListValue 和 ListPointer 这两个列表合起来就建立了一个链表。

现在运用刚刚讲到的链表元素之间的逻辑关系，来完成输出链表的程序。

因为要多次重复相似的操作，所以要使用循环。

```
01   while next != -1:              #next 是指向下一个元素的指针，它不等于-1 代表后面还有元素
02      print(ListValue[next])      #输出下一个元素中存储的值
03      next = List[Pointer[next]  #把指针变为下一个元素中存储的指针
```

完成了这一步，就可以在前面再加上其他语句了，如代码 2.3 所示。

代码 2.3　输出一个由两个列表组成的单链表

```
01   ListValue = [1,5,6,2,4,3]
02   ListPointer = [3,2,-1,5,1,4]
03   head = 0                       #head 是指向链表第一个元素的指针，需要自己定义
04   print(ListValue[head])         #直接输出第一个元素的值
05   next = ListPointer[head]       #给 next 赋初始值
06   while next != -1:              #next 是指向下一个元素的指针，不等于-1 代表后面还有元素
07      print(ListValue[next])      #输出下一个元素中存储的值
08      next = ListPointer[next]    #把指针变为下一个元素中存储的指针
```

在进入循环之前，要先把第一个元素输出。指针 next 的初始值是链表中第一个元素所存储的指针，也就是指向第二个元素的指针，所以 while 循环第一次输出的是第二个元素。

在循环的前面直接用 head 指针（指向第一个元素的指针）输出第一个元素就可以避免这种错误。当然，我们也可以在 while 循环内部加入 if 语句来代替 while 外面的输出，但是这样反而麻烦，所以不详细讲解了。

接下来要讲解的是第二种实现模拟单链表的方法——数组中套数组。我们来看下面这一行代码：

```
LinkedList = [ [1,3], [5,2], [6,-1], [2,5], [4,1], [3,4] ]
```

大数组中的每个小数组都是链表中的一个元素。每个小数组中的第一个数是这个元素存储的值，第二个数是指向下一个元素的指针。调用大数组 LinkedList 中的第 m 个小数组中的第 n 个元素可以这样写：LinkedList[m-1][n-1] （注意下标从 0 开始）。

在这一行程序中，LinkedList[m][0]表示第 m+1 个小数组中存储的值，LinkedList[m][1]表示第 *m*+1 个小数组中存储的指针。如果记不住 0 和 1 哪个是存储哪种变量的，那么可以在程序的开头先定义两个变量来存储 0 和 1：

```
01   value = 0
```

```
02    pointer = 1
03    LinkedList = [ [1,3], [5,2], [6,-1], [2,5], [4,1], [3,4] ]
```

这样，调用的时候就可以分别用 LinkedList[m][value]和 LinkedList[m][pointer]来调用第 *m*+1 个小数组中的值和指针了。

我们来编写一下这种模拟链表的输出程序。

首先来编写 while 循环内部的程序：

```
01    while next != -1:          #next 是指向下一个元素的指针，它不等于-1 代表后面还有元素
02       print(LinkedList[next][value])      #输出下一个元素中存储的值
03       next = LinkedList[next][pointer]    #把指针变为下一个元素中存储的指针
```

最后我们再把 while 循环前面的语句补上就得到了完整的程序，如代码 2.4 所示。

代码 2.4　输出一个列表套列表组成的单链表

```
01    value = 0                  #预先设置好值和指针在小数组中的位置
02    pointer = 1
03    LinkedList = [ [1,3], [5,2], [6,-1], [2,5], [4,1], [3,4] ]
04    head = 0
05    print(LinkedList[head][value])     #直接输出第一个元素的值
06    next = LinkedList[head][next]      #给 next 赋初始值
07
08    while next != -1:              #next 是指向下一个元素的指针，它不等于-1 代表后面还有元素
09       print(LinkedList[next][value])      #输出下一个元素中存储的值
10       next = LinkedList[next][pointer]    #把指针变为下一个元素中存储的指针
```

代码 2.4 和代码 2.3 的结构十分相似。除了调用元素中的值和指针的方式不同之外，其他基本一样。现在，你弄懂链表的连接方式了吗？

2.3.3　建立双链表

2.3.2 节中讲到了单链表，而在这一节中，我们要讲到一种更复杂的链表：双链表。我们知道，单链表中的每个元素是由它的值和一个指向下一个元素的指针组成的，那么，相似的，双链表中的每个元素是由它的值和两个指针组成的，一个指针指向下一个元素，一个指针指向上一个元素，如图 2.17 所示。

如图 2.18 所示，这是一个双链表的示例。双链表比起单链表的好处就是它可以双向遍历。也就是说，除了使用从第一个元素到最后一个元素的顺序来遍历之外，还可以从最后一个元素到第一个元素进行遍历。由于单链表中的每个元素只能单向运行，所以如果程序对链表遍历到第 *n* 个数时，就无法再返回第 *n* −1 个数或者之前的数了。而双链表不同，无论遍历到第几个数，

都可以通过指向前面元素的指针返回到前面。

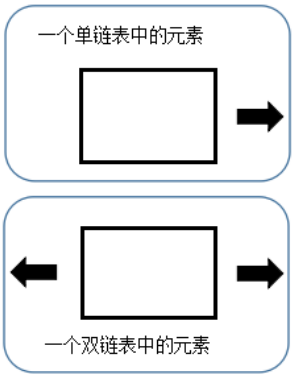


图2.17　单链表和双链表中的元素

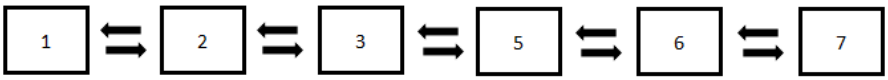


图2.18　双链表

双链表的排列方式和单链表其实很相似，只是两个元素之间又多了一个指针而已。现在就来建立一个双链表。与单链表一样，双链表也有两种模拟方法，一种是用多个分开的数组来模拟，一种是用数组套数组来模拟。

先来看第一种模拟方法：

```
ListValue = [1,5,6,2,7,3]
ListRight = [3,2,4,5,-1,1]          #用于存储下一个元素下标的指针
ListLeft = [-1,5,1,0,2,3]           #用于存储上一个元素下标的指针
```

然后是第二种模拟方法：

```
right = 1                           #小数组的第二个数存储下一个元素的下标
left = 2                            #小数组的第三个数存储上一个元素的下标
value = 0                           #小数组的第一个数存储元素的值
LinkedList = [[1,3,-1],[5,2,5],[6,4,1],[2,5,0],[7,-1,2],[3,1,3]]
```

由于在 2.3.2 节讲解单链表时说明过这么做的理由，在此就不做详细解释了。接下来编写把两种模拟双链表正向输出的程序，如代码 2.5 所示。实际上，这一段程序中由于不需要用到指向上一个元素的指针，所以和单链表的输出程序基本是一样的。

代码 2.5　正序输出双链表

```
ListValue = [1, 5, 6, 2, 7, 3]          #第一种模拟双链表的方法
ListRight = [3, 2, 4, 5, -1, 1]         #分别使用三个数组来存储数值、
ListLeft = [-1, 5, 1, 0, 2, 3]          #指向上一个元素和下一个元素的指针

head = ListLeft.index(-1)               #头指针的值为-1 在 ListLeft 中的位置
print(ListValue[head])
Next = ListRight[head]

while Next > -1:
    print(ListValue[Next])
    Next = ListRight[Next]
print()                                 #换行隔开两个链表的输出结果
right = 1                               #第二种模拟双链表的方法
left = 2
value = 0
LinkedList = [[1, 3, -1], [5, 2, 5], [6, 4, 1], [2, 5, 0], [7, -1, 2], [3, 1, 3]]
head = 0                                #提前设置头指针指向的位置
print(LinkedList[head][value])
Next = LinkedList[head][right]

while Next > -1:
    print(LinkedList[Next][value])
    Next = LinkedList[Next][right]
```

在第一种模拟方法中，可以在 ListLeft 中寻找-1 的位置从而确定链表的第一个元素。因为与单链表不同，双链表的第一个元素有一个指向它前一个元素的指针，这个指针为空，可以通过这个空指针来判断它是第一个元素。当然，在第二种模拟方法中也可以用这种方法，但是需要一个多余的 for 循环。

注意，在第一种模拟方法中，ListValue[n]、ListRight[n]和 ListLeft[n]这三个值共同组成了双链表中的一个元素。它们的顺序和第二种模拟方法的小数组中每个值的顺序是一样的。由于在图示中下一个元素一般在右侧，上一个元素一般在左侧，所以给指向下一个和上一个元素的指针取名叫 right 和 left。

2.3.4　双向输出双链表

前面我们介绍过，双链表由于每个元素有指向前后元素的两个指针，所以它可以被双向输出。这一节我们就来实现这样的双向输出。先把双链表按从第一个元素到最后一个元素的顺序输出一遍，再把它从最后一个元素到第一个元素输出一遍。

由于和这段程序有关的知识在前面都讲解过，所以自己先思考一下该怎么实现这个程序再来看下面的代码 2.6 吧！

代码 2.6　双向输出双链表

```
ListValue = [1, 5, 6, 2, 7, 3]        #和 2.3.2 节一模一样的正向输出
ListRight = [3, 2, 4, 5, -1, 1]
ListLeft = [-1, 5, 1, 0, 2, 3]

head = ListLeft.index(-1)
print(ListValue[head])
Next = ListRight[head]

while Next > -1:
    print(ListValue[Next])
    Next = ListRight[Next]

head = ListRight.index(-1)           #从这里开始的是反向输出
print(ListValue[head])
Next = ListLeft[head]

while Next > -1:
    print(ListValue[Next])
    Next = ListLeft[Next]
```

注意：本章接下来的内容将会统一采用多个数组共同组成一个链表的模拟方法。

可以发现，在反向输出的代码块中，我们所做的只是把 ListRight 和 ListLeft 互换了。思考一下双链表前后对称的特性，想想为什么会出现这种情况。

2.3.5　向单链表中添加元素

在本章剩下的部分中，我们将会学习怎样对列表中的元素进行操作。这一节中会讲到如何向单链表中添加元素。

图 2.19 所示的是一个单链表。我们想把新元素 4 插到 3 的后面、5 的前面。这时候该怎么做呢？

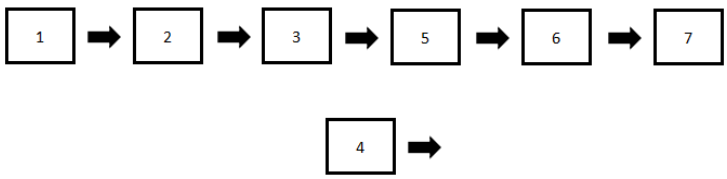


图2.19　向单链表中插入元素

首先，把新元素的指针指向它将要插入的位置后方的元素，也就是 5，如图 2.20 所示。

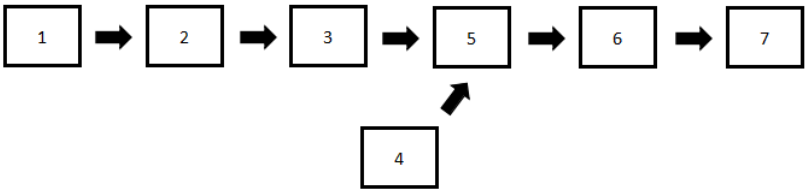


图2.20　将元素插入单链表的第一步

随后，像图 2.21 这样，把新元素前面的元素的指针指向这个新元素，在这个链表中这个元素是 3。

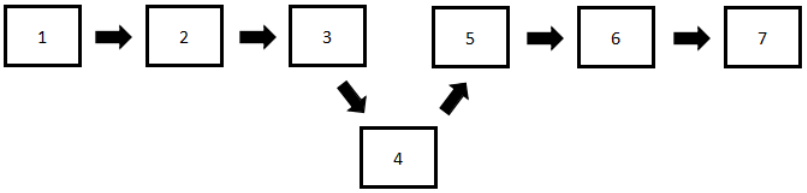


图2.21　将元素插入单链表的第二步

这样，原来的链表和新元素就被连在一起形成一个更长的链表了。你可能会问，为什么要把新元素的指针先指向它后面的元素，再让它前面的元素指向它而不是先让它前面的元素指向它呢？自己思考一下这个问题再来看答案吧！

注意：指针是存储下一个元素位置的变量。

插入元素时，已知的是新元素和它前面的元素的位置以及它们指向的位置。通过前一个元素的指针存储的位置，我们才能找到下一个元素的位置，从而和它连接上。如果在用新元素的指针存储下一个元素的位置之前，就把上一个元素的指针指向新元素，就丢失了下一个元素的位置。这样的话，整个链表就断开了。如图 2.22 所示，你会更直观地感受这个问题。

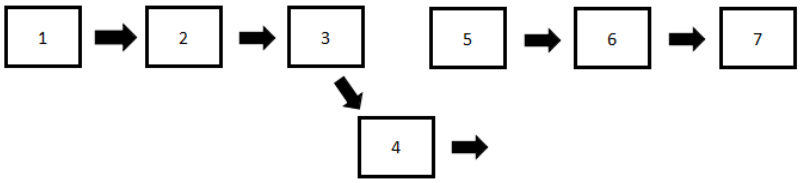


图2.22　插入单链表时不按顺序改变指针的值

了解了把元素插入单链表的方法和可能出现的问题之后，我们来写一下这个程序。

首先要定义好被插入的单链表并确定头指针的位置：

```
ListValue = [1, 5, 6, 2, 7, 3]
ListRight = [3, 2, 4, 5, -1, 1]
head = 0
```

然后用程序实现插入：

```
ListValue = [1, 5, 6, 2, 7, 3]
ListRight = [3, 2, 4, 5, -1, 1]
head = 0
prepos = 5                              #已知要插入的位置的上一个元素的位置

ListValue.append(4)                     #向数组末尾加上新元素的值 4
ListRight.append(ListRight[prepos])     #加上新元素指针指向的位置（下一个元素）
ListRight[prepos] = len(ListValue)-1    #上一个元素的指针指向新元素
```

由于列表中元素的排列顺序只取决于指针的顺序，而与它实际在数组中的排序无关，所以直接把新元素的值加在存储值的数组末尾即可，它是最后一个值，所以它的下标即为len(ListValue)−1。

但是，在把上一个元素的指针指向新元素之前，要把新元素的指针指向下一个元素。下一个元素的位置现在存储在上一个元素的指针中，也就是ListRight数组中下标为prepos的变量中。把这个指针的值加在 ListRight 的最后，就刚好和新添加到 ListValue 中的值对应了。最后再把上一个元素的指针赋值为新元素的位置，整个插入就完成了。

接下来在这个程序中添加输出链表的代码，以验证插入是否成功，如代码 2.7 所示。

代码 2.7　向单链表中插入元素

```
def Output(ListValue,ListRight,head):  #定义一个函数用于输出链表
    print(ListValue[head])
    next = ListRight[head]
    while next != -1:
        print(ListValue[next])
        next = ListRight[next]

ListValue = [1, 5, 6, 2, 7, 3]
ListRight = [3, 2, 4, 5, -1, 1]
head = 0
prepos = 5                              #已知要插入的位置的上一个元素的位置

Output(ListValue,ListRight,head)        #输出未插入元素的链表
print()
```

```
15
16    ListValue.append(4)                       #向数组末尾加上新元素的值 4
17    ListRight.append(ListRight[prepos])       #加上新元素指针指向的位置（下一个元素）
18    ListRight[prepos] = len(ListValue)-1      #上一个元素的指针指向新元素
19
20    Output(ListValue,ListRight,head)          #输出已经插入元素的链表
```

由于这段程序中多次使用了相似的功能，所以定义函数能减少很多步骤。如果你对函数的使用还不太熟悉的话，可以复习一下第 1 章的内容。在这段程序中，在元素被插入前和被插入后分别输出一次链表，中间用空行隔开，可验证我们的程序是否正确。

2.3.6　向双链表中添加元素

讲完了如何向单链表中添加元素，本节要来讲解如何向双链表中添加元素。我们先来看一看整个过程的图示。

图 2.23 所示的是一个未插入新元素的双链表，图 2.24 所示的是一个要被插入的元素。

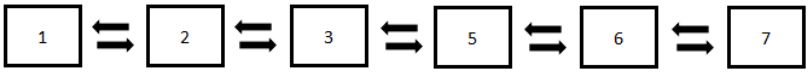


图2.23　向双链表中插入元素

图2.24　一个要被插入的元素

接下来进行第一步操作：给新元素指向前后元素的指针赋值。这两个指针的赋值顺序可以调换，不影响结果，图 2.25 所示的是直接一次性把两个指针都加上。

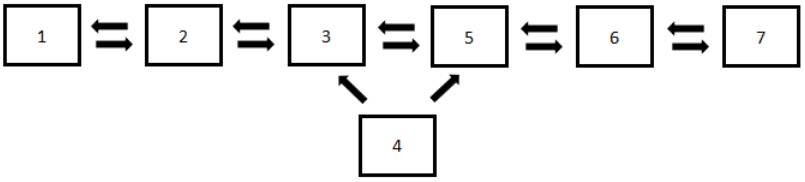


图2.25　把元素插入双链表的第一步

接下来，为前后两个元素指向新元素的指针赋值，如图 2.26 所示，整个插入过程就结束了。同样，由于新元素的位置已知，所以这两个指针的赋值顺序可以调换，不影响结果。

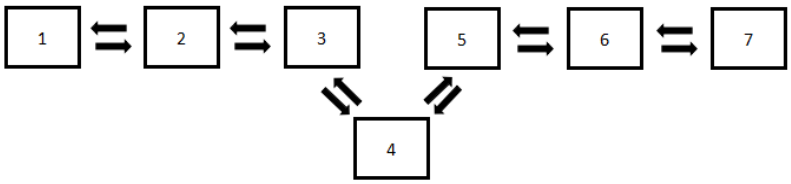


图2.26　把元素插入双链表的第二步

和单链表类似，如果插入的这两步互换顺序的话，由于我们没有存储后一个元素的位置，链表会断开。

双链表比起单链表的好处就是可以通过知道上一个元素的位置来插入新元素，也可以通过知道下一个元素的位置来插入新元素，这大大方便了查找。单链表无法回到上一个元素的特性给查找带来了很多麻烦，所以虽然双指针链表比单指针链表要复杂一些，但它也更常用。

由于这个过程只是在代码 2.7 的基础上做了一些改动和添加，所以先自己思考一下该怎么解决这个问题，再来看完整的代码，如代码 2.8 所示。

代码 2.8　向双链表中插入元素

```
01   def Output(ListValue,ListRight,head):              #定义一个函数用于输出链表
02       print(ListValue[head])
03       next = ListRight[head
04       while next != -1:
05           print(ListValue[next])
06           next = ListRight[next]
07
08   ListValue = [1, 5, 6, 2, 7, 3]                     #建立一个双链表
09   ListRight = [3, 2, 4, 5, -1, 1]
10   ListLeft = [-1, 5, 1, 0, 2, 3]
11   head = 0                                           #确定第一个元素的位置
12   prepos = 5                                         #确定要插入的位置的前一个元素的位置
13
14   Output(ListValue,ListRight,head)                   #输出未经插入的链表
15   print()
16
17   ListValue.append(4)                                #在 ListValue 数组末尾加上新元素的值
18   ListRight.append(ListRight[prepos])                #给新元素的两个指针赋值
19   ListLeft.append(prepos)
20   ListLeft[ListRight[prepos]] = len(ListValue)-1 #将前后元素的指针指向新元素
21   ListRight[prepos] = len(ListValue)-1
22
23   Output(ListValue,ListRight,head)                   #输出插入后的链表
```

这段程序比起代码 2.7，多了第 19 和 20 两行给指向前一个数的指针赋值的程序（ListLeft）。其中第 19 行的程序是把新元素的 left 指针指向前一个数，也就是我们已知的 prepos。第 20 行的程序是把下一个元素的 left 指针指向新元素。ListRight[prepos]是下一个元素的位置，要通过这个位置来调用下一个元素的 left 指针，也就是 ListLeft[ListRight[prepos]。

仔细观察你会发现，输出函数和代码 2.7 中的一模一样，这是因为输出一个链表不需要用到双向的指针。如果你觉得这样没法证明插入是正确的，可以把 ListValue、ListRight 和 ListLeft 这三个数组输出以进行验证。

用相似的逻辑，自己思考一下，在双链表中，如果只知道插入位置的下一个元素的位置，该如何插入新的元素呢？

2.3.7　删除列表中的元素

讲完如何向链表中插入元素，接下来的内容是如何删除链表中的元素。

我们来看一看删除单链表中的元素的图示。

图 2.27 所示的是原来的单链表，图 2.28 所示的是完成了单链表中元素的删除。我们发现，单链表的删除只需要一步就可以完成。直接通过当前元素指向下一个元素的指针来找到下下一个元素的位置，然后把这个位置赋给当前元素的指针，这样当前元素的指针就跳过下一个元素，直接连接了下下一个元素。所以，即使原来的下一个元素仍然有指向下下一个元素的指针，以指针的顺序遍历列表时，这个元素也不会出现了。

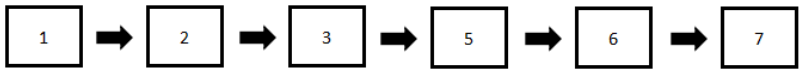


图2.27　未删除元素的单链表

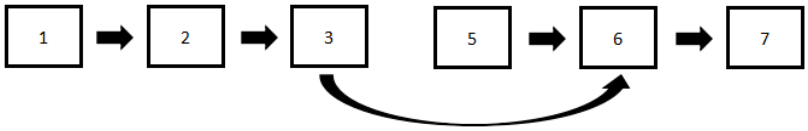


图2.28　删除了元素的单链表

现在来编程实现这个删除过程。

第一步还是建立这个单链表并确定头指针的位置：

```
01    ListValue = [1, 5, 6, 2, 7, 3]                    #建立单链表
```

```
02    ListRight = [3, 2, 4, 5, -1, 1]
03    head = 0                                          #确定头指针
```

接下来就是删除元素的过程。我们想删除的是值为 5 的元素，它在 ListValue 中的位置是 1，需要知道前一个元素的下标，也就是 3 的下标，是 5。真正删除元素的代码只需要一行。我们通过 ListRight[prepos]得：

```
01    ListValue = [1, 5, 6, 2, 7, 3]                    #建立单链表
02    ListRight = [3, 2, 4, 5, -1, 1]
03    head = 0                                          #确定头指针
04    prepos = 5                                        #确定要删除的元素的前一个数的位置
05    ListRight[prepos] = ListRight[ListRight[prepos]]  #删除元素
```

到下一个元素的位置，再通过这个位置来得到下下一个元素的位置，也就是 ListRight[ListRight [prepos]]。把得到的这个位置赋给要删除的元素的前一个数，也就是当前的数的指针，就完成了整个删除过程。

最后，加上输出，就得到了完整程序，如代码 2.9 所示。

代码 2.9　删除单链表中的元素

```
01    def Output(ListValue,ListRight,head):             #输出函数
02        print(ListValue[head])
03        n = ListRight[head]
04        while n != -1:
05            print(ListValue[n])
06            n = ListRight[n]
07
08    ListValue = [1, 5, 6, 2, 7, 3]                    #建立单链表
09    ListRight = [3, 2, 4, 5, -1, 1]
10    head = 0                                          #确定头指针
11    prepos = 5                                        #确定要删除的元素的前一个数的位置
12
13    Output(ListValue, ListRight, head)                #输出删除前的链表
14    print()
15
16    ListRight[prepos] = ListRight[ListRight[prepos]]        #删除元素
17
18    Output(ListValue, ListRight, head)                #输出删除后的链表
```

接下来试着删除双链表中的元素。图 2.29 所示的是未删除元素的双链表，而图 2.30 所示的是删除完成后的链表。和前面一样，我们知道要删除的元素的前一个数的位置。然后通过要删除的元素来确定下下一个元素的位置，最后直接连接要删除的元素的前后两个数就完成了删除。虽然被删除的元素仍然有指针指向前后两个元素，但它不会出现在链表中。

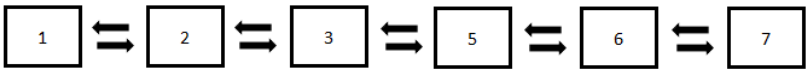


图2.29　未删除元素的双链表

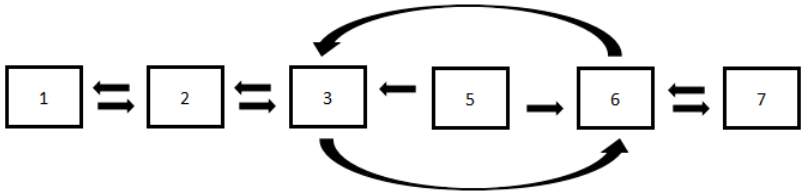


图2.30　删除了元素的双链表

接下来要用程序实现这个过程。

首先还是建立双链表、确定头指针和要删除的元素的前一个元素，然后再对链表进行操作。

```
ListValue = [1, 5, 6, 2, 7, 3]                    #建立双链表
ListRight = [3, 2, 4, 5, -1, 1]
ListLeft = [-1, 5, 1, 0, 2, 3]
head = 0                                          #确定头指针
prepos = 5                                        #确定前一个元素的位置
```

现在我们对链表进行删除元素的操作。

```
01   ListValue = [1, 5, 6, 2, 7, 3]          #建立双链表
02   ListRight = [3, 2, 4, 5, -1, 1]
03   ListLeft = [-1, 5, 1, 0, 2, 3]
04   head = 0                                 #确定头指针
05   prepos = 5                               #确定前一个元素的位置
06
07   ListRight[prepos] = ListRight[ListRight[prepos]] #把前一个元素的 right 指针指向后一个元素
08   ListLeft[ListRight[ListRight[prepos]]] = prepos #把后一个元素的 left 指针指向前一个元素
```

其中，第 7 行的程序和我们前面讲到的从单链表中删除元素的程序是一样的，就是把要删除的元素的前一个元素的 right 指针指向下下一个元素。第 8 行的程序用来把要删除的元素的后一个元素的 left 指针指向当前元素。

ListRight[prepos]存储的是要删除的元素的位置，则 ListRight[ListRight[prepos]]存储的是要删除的元素的下一个元素的位置，所以 ListLeft[ListRight[ListRight[prepos]]]就是这个元素的 left 指针，把当前元素的位置赋给它，也就是 prepos，删除就完成了。虽然这一步的逻辑有一点绕，但是多练习之后就适应了。

接下来，像前面的程序那样加上输出函数就可以验证结果了。

考虑到双链表的特性，再自己思考一下，如果只知道要删除的元素的后一个数，该怎么删除这个元素呢？

关于指针、双指针和链表的学习到此就告一段落了，接下来请继续学习其他的算法吧！

第 3 章

哈希算法

哈希算法又称散列函数算法，是一种查找算法。简单来说，就是把一些复杂的数据，通过某种函数映射关系，映射成更加易于查找的方式。但是这种映射关系有可能会发生多个关键字映射到同一地址的现象，我们称之为冲突。在这种特殊情况下，需要对关键字进行第二次或更多次的处理，在其他的大多数情况下，哈希算法可以实现在常数时间内存储和查找这些关键字。本章主要涉及的知识点和问题有如下几项。

- 什么是哈希：简单介绍什么是哈希及哈希的原理。
- 两个数的和：快速寻找两个数的和。
- 单词模式匹配：简单的模式匹配问题。

3.1 什么是哈希

随着信息技术的迅猛发展，信息的重要性不言而喻，我们常常需要使用搜索引擎来解决日常生活中的大量问题，其中的一项关键技术就是数据查找算法。数据查找已经成为人们日常生活的一部分，因此，如何提高数据查找的效率是软件工程师不断研究的课题。

常见的数据查找算法一般可分为如下几种，如图 3.1 所示。

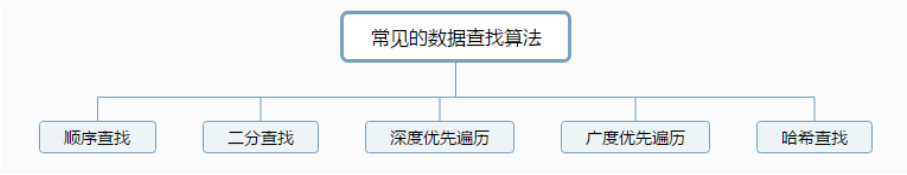


图3.1　常见的数据查找算法

- 顺序查找是最简单的查找方式，需要对数据集中的数据逐个进行匹配，所以效率相对较低，不太适合大数据量的查找问题。
- 二分查找的查找效率很高，但是要求数据必须有序，而对数据排序通常需要更多的时间开销。
- 深度优先遍历、广度优先遍历对于大数据量的查找问题效率并不高。
- 哈希查找算法由于其查找速度快，查询、插入、删除操作简单等原因而获得了广泛的应用。很多问题本质上都是查找问题。解决查找问题，哈希算法是较好的选择。

哈希算法（Hash）是什么呢，哈希算法进行查找的基本原理是根据数据量预先设置一个长度为 M 的数组，使用一个哈希函数 F 并以数据的关键字作为自变量，得到唯一的返回值，返回值的范围为 $0 \sim M-1$，这样就可以利用哈希函数 F 将数据元素映射到数组的某一位下标并把数据存放在对应位置上。查找时，利用哈希函数 F 计算该数据应该存储在哪里，再到相应的存储位置取出查找的数据。

例如，有一组数据 3、11、8、6 需要存储，哈希函数为 $F(x)=x\%7$，那么利用哈希函数 F 存储的数据如表 3.1 所示。

表 3.1　哈希函数示例

	8		**3**	**11**		**6**

按照统一的哈希函数存储数据，也按照统一的哈希函数查找数据，按照这种方法查找数据，理论上可以达到常数级别的查找效率。

就像在生活中，要想时刻能够找到自己的东西，最好的办法就是把东西放到固定的地方，每次需要它的时候就去相应的地方找，用完以后再放回原处。哈希是一种高效的存储算法，也是一种高效查找算法。

哈希就像一本字典，当你需要查词的时候，通过目录找到页码，再到对应页码就能找到所需要的内容了。

一般情况下，哈希算法的查询效率可以达到常数级别，哈希表成为直接寻址的有效替代方

案。然而由于关键字的取值可能在一个很大的范围，数据在通过哈希函数进行映射的时候，很难找到一个哈希函数，使得这些关键字不能映射到唯一的值，就会出现多个关键字映射到同一个值的现象，这种现象我们称为冲突。

这就对哈希函数的设计提出了更高的要求，如果哈希函数设计得不好，查询效率可能会降低为顺序查找的效率。找到绝对完美的、没有任何冲突的哈希算法也是较困难的，对于特定的数据集，好的哈希算法也是少之又少，因此如果想高效地解决哈希冲突，降低解决哈希冲突时的查询长度也是一个要重点考虑的方面。

解决哈希冲突的方法有很多，如链地址法、二次再散列法、线性探测再散列、建立一个公共溢出区等方法。

本章主要介绍哈希算法的应用，因此只对链地址法进行讲解。链地址法处理冲突的方法本质上是一种数组加链表的处理方法，当发生多个数据通过哈希函数映射后得到相同的哈希值时，通常把具有相同哈希地址的关键字放在同一个链表中，称该链表为同义词链表，该链表也被称为桶。

当把相同哈希地址的关键字数据都放在同一个链表中时，有 N 个哈希地址就有 N 个链表。同时，用数组 Hash[0..N−1]存放每个链表的头指针，之后把哈希地址为 i 的数据全部以节点的方式插入对应的链表里，如图 3.2 所示。

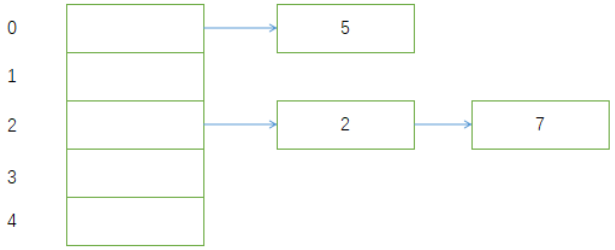


图3.2　链地址法处理冲突

注意：链地址法本质上是数组+链表的数据结构。

链地址法存储数据的过程是这样的，首先建立一个数组 Hash 存储所有链表的头指针，由数据的关键字 Key 通过对应的哈希函数计算出哈希地址，找到相应的桶号，之后建立新的节点存储该数据，并把节点放到桶内链表的最后面或者最前面。

和存储数据的方法类似，查找数据的时候，也是由数据的关键字通过哈希函数计算关键字对应的哈希地址，之后顺序比较桶的内部节点是否与所查找的关键字一样直到找到数据为止。如果全部节点都不和关键字一样，则说明哈希表里没有该数据。这个解决冲突的方法对哈希函

数的要求很高，如果哈希函数选得不太好的话，哈希表的查找效率会退化为链表的查找，也就是顺序查找。

用链地址法构造的散列表，插入和删除节点操作易于实现，所以构造链表的时间开销很低，但是指针需要开辟额外的地址空间，当数据量很大时，会扩大哈希表规模，内存空间需求较大。

从上面的分析可以看到，构造优秀的哈希函数和选择解决冲突的方法是哈希查找算法的关键。不管多么高明的算法都不可能避免冲突问题，但是构建计算简单、高效快速，并且能够将关键字集合均匀地分布在地址集中，使得冲突达到最小，一直是计算机科学家追求的目标。

3.2 两个数的和

数学课上，老师出了一道题目，要求在给定的一些数字中找出两个数，使得它们的和为 *N*，前提是这些数据中保证有答案，并且只有一个答案。例如给定 5 个数字：3、4、5、7、10，从中选择两个数使它们的和为 11，可以选择 4 和 7，如图 3.3 所示，这个问题该如何解决呢？

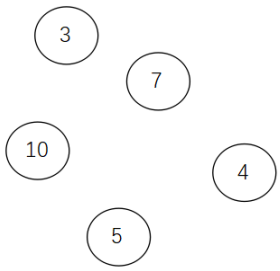


图3.3　链地址法处理冲突

3.2.1 问题求解 1

在上面的例子中，可以把数据放到一个数组中，并设置目标值 target，问题就变成了在数组中寻找两个数使得它们的和为目标数，并返回它们的编号，还要注意编号不是下标，而是数组的下标+1。

先来看第一个办法，可以使用第 2 章中介绍的双指针的方法来解决这个问题。为了找到两个元素的和为目标值，先将数组从小到大排序。由于最后要得到的是两个数的原有编号，而不是两个数本身，因此需要保留原有数据的下标，那么，排序时就需要把数据复制到一个新的数组中，然后对新的数组进行排序。代码如下：

```
01 def twoSum(nums, target):
02   res= []                                #存放结果编号数据
```

```
03   newnums= nums[:]                    #深拷贝，把原数据复制到 newnums 里
04   newnums.sort()                      #对新数组排序
05   #查询两个数使得它们的和为目标值 target
06   return (res[0]+1,res[1]+1)          #返回结果集
```

注意：返回值是数据的编号，即原始数据的下标+1。

数组排序后，就可以开始查找了。建立两个指针 left 和 right，分别指向新数组的第一个元素和最后一个元素。如果两个指针指向的两个数据相加的和等于目标值，那么查找结束，返回这两个数的下标；如果它们的和小于目标值，则说明 left 指向的数据太小了，需要让 left 指针向右移动一位；如果它们的和大于目标值，这说明 right 指向的数据太大了，需要让 right 指针向左移动一位。重复这个过程，直到找到答案。查找过程如图 3.4、图 3.5 和图 3.6 所示。

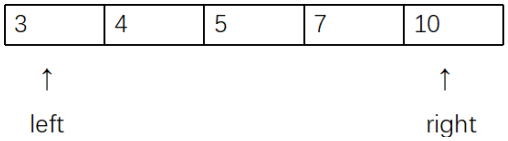


图3.4　初始状态

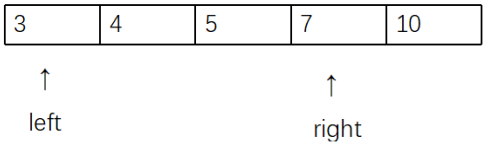


图3.5　数据和大了移动右指针

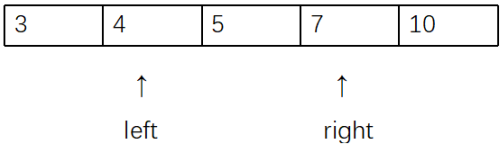


图3.6　数据和小了移动左指针

由于 left 一直向右移动，right 一直向左移动，总有一个时刻 left 和 right 会重合甚至 left 比 right 还要大，那么这说明数组中没有两个数的和是目标值。由于数据里一定存在两个数的和为目标值，因此这种情况并不会出现。代码如下：

```
01 def twoSum(nums, target):
02   res= []                             #存放结果编号数据
03   newnums= nums[:]                    #深拷贝，把原数据复制到 newnums 里
04   newnums.sort()                      #对新数组排序
```

```
    left = 0
    right = len(newnums) - 1   #定义 left 和 right 指针分别指向新数组的开头和结尾
    while left < right:
        if newnums[left] + newnums[right] == target:     #寻找这两个数的下标

         elif newnums[left] + newnums[right] < target:
              left = left + 1                            #让 left 指针向右移动一位
         elif newnums[left] + newnums[right] > target:
              right = right - 1                          #让 right 指针向左移动一位
    return (res[0]+1,res[1]+1)                           #返回结果集
```

使用这种方法找到的两个元素的下标是新数组的下标，并不是原始数据的下标。为了得到想要的答案，还需要在原始数组中查找这两个元素的位置，并存储到 res 数组中。来看一下最终代码吧。

3.2.2 解法 1 的最终代码

解法 1 的最终代码如代码 3.1 所示。

代码 3.1 两个数的和的解法 1 代码

```
def twoSum(nums, target):
    res= []                     #存放结果编号数据
    newnums= nums[:]            #深拷贝，把原数据复制到 newnums 里
    newnums.sort()              #对新数组排序
    left = 0
    right = len(newnums) - 1    #定义 left 和 right 指针分别指向新数组的开头和结尾
    while left < right:
        if newnums[left] + newnums[right] == target:
            for i in range(0,len(nums)):         #在原始数组中寻找第一个元素的原始下标
                if nums[i] == newnums[left]:
                    res.append(i)                #将下标加入结果集
                    break
                for i in range(len(nums)-1,-1,-1):   #在原始数组中寻找第二个元素的原始下标
                 if nums[i] == newnums[right]:
                    res.append(i)                #将下标加入结果集
                    break
            res.sort()
            break
         elif newnums[left] + newnums[right] < target:
              left = left + 1                    #让 left 指针向右移动一位
         elif newnums[left] + newnums[right] > target:
              right = right - 1                  #让 right 指针向左移动一位
    return (res[0]+1,res[1]+1)#返回结果集
```

这种算法的前提是要先进行一次排序，查找效率不是很高，如何才能提高查找效率呢，是不是有更好的算法呢？

3.2.3　问题求解 2

在第一种解法的基础上，我们接下来研究更加高效的算法，来看一下哈希函数是怎么帮助解决这个问题的。

首先，看一下在双指针解法中，时间都浪费在了哪里。双指针算法的前提是数据有序，所以解法 1 的第一步是对数据排序，即便是使用最优秀的排序算法也是比较耗时的。接下来，为了寻找答案，左指针和右指针在不断移动，当左右指针发现了答案时，由于需要返回原始数据的位置，还要去原始数组中去寻找这两个数据的位置。这两处操作都是比较耗时的，而且没有必要。

那么，有什么办法不需要排序数据，而且不需要寻找原始位置呢？答案就是哈希算法了。

为了使用哈希算法，我们需要先建立一个字典，用于存放数据和下标的对应关系，如下所示。

注意：字典是一个键值对的集合。

```
01   def twoSum(num, target):
02     dict = {}              #建立一个字典，存储数据和下标的对应关系
```

那么，这个字典是如何帮助我们解决问题的呢？本题是寻找两个数使得它们的和为目标数，每当给定一个数 *m*，其实问题就变成了数据集合中是否有一个数是 target−m，可以通过使用字典记录目前已经出现过哪些数字，这样每次出现一个新的数字时，就去字典中查找有没有对应的数字，如果有则说明找到了，没有的话就把该数放到字典中去，以备之后查询使用。

```
01 i=#someValue                                  #i 为给定的某个值
02 m=nums[i]                                     #定义 m 为当前待查询数字
03 if target-m in dict:                          #判定 target-m 是否已经在字典中
04    return (dict[target-m]+1, i+1)             #如果已经存在，则返回这两个数的下标
05 dict[m] = i                                   #如果不存在则记录键值对
```

注意：字典使用 dict[key] = value 来记录键值对的关系。

使用字典记录数据，再去字典中查找数据的方式就是我们所说的哈希方法，使用统一的哈希函数把数据对存储到字典中，再使用统一的哈希函数从字典中把数据取出来，理论上是可以达到常数级别的查询速度的。

让我们来看一下最终代码吧。

3.2.4 解法 2 的最终代码

解法 2 的最终代码如代码 3.2 所示。

代码 3.2 两个数的和的解法 2 代码

```
01 def twoSum(nums, target):
02       dict = {}
03       for i in range(len(nums)):
04         m=nums[i]                                  #定义 m 为当前待查询的数字
05         if target-m in dict:                       #判定 target-m 是否已经在字典中
06                return (dict[target-m]+1, i+1) #如果已经存在，则返回这两个数的下标
07         dict[m] = i                                #如果不存在则记录键值对
```

显而易见，使用哈希算法解决查找问题，不仅效率高、代码少而且容易理解，日后遇到查询问题能用哈希算法的时候，一定要记得使用哈希算法。本问题还有一些后续问题，比如求三个数的和以及求四个数的和的问题，有兴趣的读者可以继续研究这些问题。

3.3 单词模式匹配

模式匹配问题是一个经典问题，先研究一下简单的单词模式匹配问题。首先给定两个字符串，一个是单词模式字符串，另一个是目标字符串。之后检查目标字符串是否为给定的单词模式，即求目标字符串中单词出现的规律是否符合单词模式字符串中的规律。

例如，单词模式字符串为“一二二一”，目标字符串为“苹果 香蕉 香蕉 苹果”，二者的规律一样，匹配成功，如图 3.7 所示。

假如单词模式字符串不变，目标字符串变为“香蕉 苹果 香蕉 苹果”，则二者的规律不一样，匹配不成功，如图 3.8 所示。

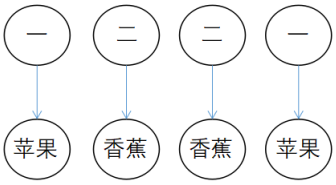


图3.7 单词字符串匹配成功

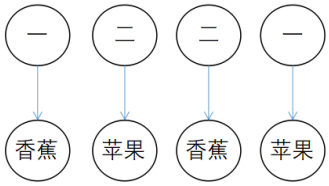


图3.8 单词字符串匹配不成功

同理，假如目标字符串变为“香蕉 香蕉 香蕉 香蕉”，二者的规律也是不一样的，匹配不成功，如图 3.9 所示。

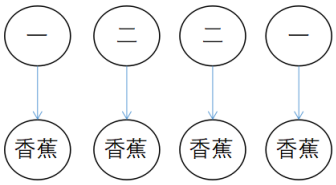


图3.9　单词字符串匹配不成功

3.3.1　问题求解

我们来分析一下这三个例子，在第一个例子中，“一”对应“苹果”，“二”对应“香蕉”，除此之外再无其他映射关系，如图 3.10 所示。

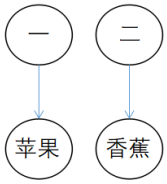


图3.10　单词字符串一对一映射关系

在第二个例子中，第一个位置的“一”对应了“香蕉”，第四个位置的“一”对应了“苹果”，而第三个位置的“二”也对应了“香蕉”，这里出现了“一”、“二”对应多个字符串的情况，如图 3.11 所示。

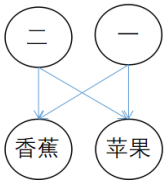


图3.11　单词字符串多对多映射关系

注意：映射关系分为一对一关系、一对多关系和多对多关系。

再来看第三个例子，第一个位置的“一”对应了“香蕉”，第二个位置的“二”也对应了“香蕉”，出现了字符串一对多的情况，如图 3.12 所示。

因此，可以发现，如果单词模式字符串和目标字符串只存在一对一的对应关系，不存在一对多和多对多的对应关系，就可以说明两个字符串匹配成功。

现在的问题其实已经转换为寻找映射关系的问题，本质上也是一个查找问题。既然是查找

问题，第一个想到的就是这个问题能不能使用哈希算法。

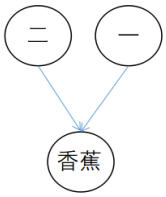


图3.12　单词字符串一对多映射关系

对于本题来说，我们建立模式字符串中每个字符和目标字符串中每个单词之间的映射关系，而哈希表本身就是一种映射关系，因此可以使用哈希算法来存储这种关系。

第一项任务是建立哈希表来存储数据，由于不仅需要排除一个模式对应多个字符串的情况，还需要排除多个模式对应一个字符串的情况，我们需要建立两个哈希表：hash 和 used。

hash 用来存储模式字符串和目标字符串的对应关系，used 记录目前已经使用的字符串，对第二个哈希表如果不明白的话，没关系，先往后看。另外，还要先排除两个字符串长度不相等的情况，来看一下初始代码吧。

```
01 def wordPattern(wordPattern, input):
02   word = input.split(' ')                    #目标字符串中的单词以空格隔开
03     if len(word) != len(wordPattern):        #如果两个字符串的长度不一样，则肯定不匹配
04       return False
05     hash = {}                                #记录模式字符串和目标字符串的对应关系
06     used= {}                                 #记录目前已经使用过的字符串都有哪些
```

首先解决第一个问题，就是模式字符串中每个字符只能对应目标字符串一个单词的问题。为此，我们每次拿到模式字符串中的一个字符的时候，需要检查一下它是否已经被记录过映射关系，如果出现过就要检查之前的映射关系和这次的单词是否一致，如果不一致则返回不成立。如果是第一次出现，那就把它存储在 hash 哈希表中。

```
01   for i in range(len(wordPattern)):
02       if wordPattern[i] in hash:           #检查模式字符串中的字符是否已经被记录过映射关系
03          if hash [wordPattern[i]] != word[i]:#不是第一次出现，则检查映射关系是否一致
04             return False
05       else:
06            hash [wordPattern[i]] = word[i]     #第一次出现，则加入哈希表
07   return True                                 #没有任何问题则返回成立
```

接下来，还需要解决第二个问题。当模式字符串中的某个字符第一次出现时，还需要判断这个单词是否已经和其他的模式字符绑定，这时需要用到 used 哈希表。每当创建一种映射关

系的时候，都需要在 used 中保存，同时还需要检查这个单词是否已经使用过，代码如下。

注意：由于查找单词是否已经被用过也是查找问题，所以也需要用一个哈希表来存储。

```
for i in range(len(wordPattern)):
    if wordPattern[i] in hash:          #检查模式字符串中的字符是否已经被记录过映射关系
       if hash [wordPattern[i]] != word[i]:#不是第一次出现，则检查映射关系是否一致
          return False
    else:
      if word[i] in used:               #检查这个单词是否已经使用过，使用过返回不成立
         return False
      hash [wordPattern[i]] = word[i]   #第一次出现，则加入哈希表
      used[word[i]] = True              #在 used 中保存哪些单词已经使用过
return True                             #没有任何问题则返回成立
```

3.3.2　最终代码

单词模式匹配问题的完整代码如代码 3.3 所示。

代码 3.3　单词模式匹配问题

```
def wordPattern(wordPattern, input):
 word = input.split(' ')                #目标字符串中的单词以空格隔开
  if len(word) != len(wordPattern):     #如果两个字符串的长度不一样，则肯定不匹配
    return False
 hash = {}                              #记录模式字符串和目标字符串的对应关系
 used= {}                               #记录目前已经使用过的字符串都有哪些
  for i in range(len(wordPattern)):
    if wordPattern[i] in hash:          #检查模式字符串中的字符是否已经被记录过映射关系
       if hash [wordPattern[i]] != word[i]:#不是第一次出现，则检查映射关系是否一致
          return False
    else:
      if word[i] in used:               #检查这个单词是否已经使用过，使用过返回不成立
         return False
      hash [wordPattern[i]] = word[i]   #第一次出现，则加入哈希表
      used[word[i]] = True              #在 used 中保存哪些单词已经使用过
return True
```

3.4　猜词游戏

小朗朗在和他的好朋友多多玩猜数字（Bulls and Cows）游戏。这个游戏的玩法是这样的：一个人写下几个数字让另外一人猜，当每次答题方猜完之后，出题方会给答题方一个提示，告

诉他刚才的猜测中有多少位数字和确切位置都猜对了（称为“Bulls”，公牛），还有多少位数字猜对了但是位置不对（称为“Cows”，奶牛）。答题方将会根据出题方的提示继续猜，直到猜出秘密数字为止。

可以保证的是，秘密数字和猜测数字的位数是一样的。

我们需要写一个程序，它能够根据秘密数字和朋友的猜测数返回提示，其中用 A 表示公牛，用 B 表示奶牛。

请注意秘密数字和朋友的猜测数字都可能含有重复数字。

例如秘密数字为 2018，猜测数字为 8021，由于 0 这个数不仅数字猜对了，位置也和秘密数字一致，所以它算一个公牛（A），而其他三位都是数字猜对了，但是位置不对，所以只能算是奶牛（B），我们的程序应该返回 1A3B，如图 3.13 所示。

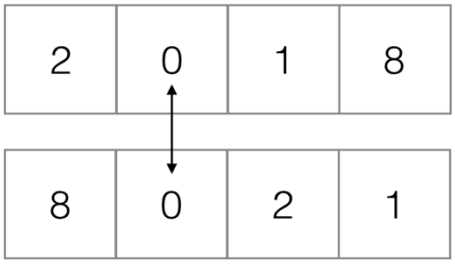


图3.13　猜词游戏例子

再比如，秘密数字为 1123，猜测数字为 9111，由于 9111 中的第一个 1 不仅数字猜对了，位置也和秘密数字一致，所以它算一个公牛（A），而其他两个 1 是数字猜对了，但是位置不对，所以只能算是奶牛（B），而且只能算一个，因为一旦秘密数字中的某一位和猜测中的某一位匹配，该数就不能和其他数字匹配了。我们的程序应该返回 1A1B，如图 3.14 和图 3.15 所示。

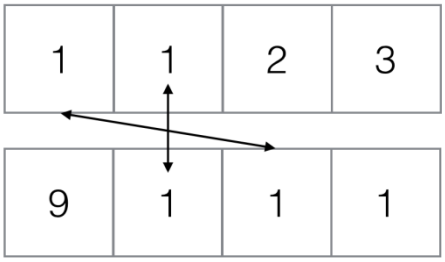


图3.14　猜词游戏例子2（1）

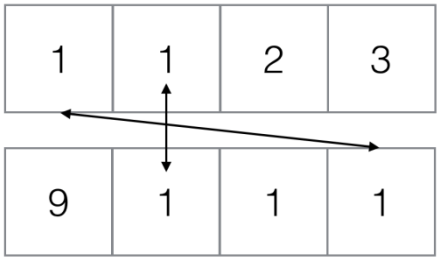


图3.15　猜词游戏例子2（2）

3.4.1　问题求解

对于这个问题，最直接的想法是把猜测数字中的每一位数字取出来在秘密数字中找一圈。如果这个数字和秘密数字中的某一位数字一样并且位置一样，则为一个公牛（A），如果仅仅只是数字一致的话，则是一个奶牛（B）。同时，为了不重复对应，还需要在每次找到对应关系后都对秘密数字已经对应的位置进行记录。

上面的办法虽然简单易行，但是对于大数据量的问题，这个算法就有点吃不消了，并且在每次寻找的过程中存在很多重复的判断。

这个问题本质上还是一个查找问题，既然是查找问题，那么首先要想到的就是哈希。在每一轮查找的过程中如果使用哈希将会大大降低查询时间，提高查找效率。

先解决第一个问题，查找公牛（A）的数量。这个问题比较简单，首先定义一个变量 A 表示公牛的数量，之后循环把每个位置上的数字都取出来，按位置对比即可，数字一样则让 A 加 1，代码如下：

```
01 def getHint( secret, guess):
02   A = 0
03   B = 0
04   for i in range(len(secret)):
05         if secret[i] == guess[i]:
06            A += 1
```

那么，如何解决计算奶牛的数量的问题呢？为了解决这个问题，我们使用哈希来加快速度，先建立两个字典：

```
01 def getHint( secret, guess):
02  secret_dict = {}
03  guess_dict = {}
```

那么，为什么需要建立两个字典呢？因为在配对的过程中需要知道哪些位置的数字配对了，哪些还没有配对。

记录每一位的情况比较麻烦，有没有什么简便方法能够简化这个过程呢？我们来观察一下最终的结果和数据的关系，对于秘密数字为 1123，猜测数字为 9111 的例子，为什么 B 的数量是 1 呢？

在秘密数字中第二个 1 已经算作公牛（A），而第一个数字 1 只能和猜测数字中两个 1 中的某一个 1 对应，也就是说，最终 B 的数量是由该数在秘密数字中和在猜测数字中更小的那方决定的。

为此，我们只需要记录秘密数字和猜测数字中未匹配的数字和它的个数，之后看相同数字的最小数即可。

那么，在分析 A 的数量的同时，记录其他未匹配数字的数量，代码如下：

```
01 def getHint( secret, guess):
02   secret dict = {}
03   guess dict = {}
04   A = 0
05   B = 0
06   for i in range(len(secret)):
07      if secret[i] == guess[i]:
08         A += 1
09      else:
10      if secret[i] in secret dict:
11           secret dict[secret[i]] = secret dict[secret[i]] + 1
12         else:
13          secret dict[secret[i]]=1
14         if guess[i] in guess dict:
15              guess dict[guess[i]] = guess dict[guess[i]] + 1
16          else:
17              guess_dict[guess[i]]=1
```

记录好了数量，接下来就可以根据两个数组中的数的最小值来判断 B 的数量了，代码如下：

```
01 for digit in secret_dict:
02      if digit in guess_dict:
03           B += min(secret_dict[digit], guess_dict[digit])
```

3.4.2 最终代码

猜词游戏的最终代码如代码 3.4 所示。

代码 3.4 猜词游戏

```
01 def getHint( secret, guess):
02      """
03      :type secret: str
04      :type guess: str
05      :rtype: str
06      """
07      secret_dict = {}
08      guess_dict = {}
09
10      A = 0
```

```
    B = 0
    for i in range(len(secret)):
        if secret[i] == guess[i]:
            A += 1
        else:
            if secret[i] in secret dict:
                secret dict[secret[i]] = secret dict[secret[i]] + 1
            else:
                secret dict[secret[i]]=1
            if guess[i] in guess dict:
                guess dict[guess[i]] = guess dict[guess[i]] + 1
            else:
                guess dict[guess[i]]=1
    for digit in secret dict:
        if digit in guess dict:
            B += min(secret dict[digit], guess dict[digit])
    return str(A)+"A"+str(B)+"B"
```

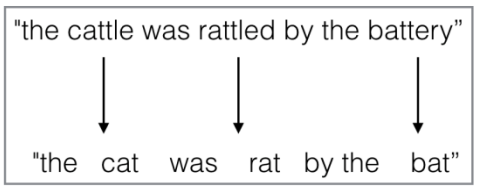
3.5　神奇的词根

在英语课上，英语老师为了让同学们掌握常见的词根来帮助理解和记忆单词，发明了一个游戏。

先来解释一下词根（root）的概念，它可以跟其他一些词组成另一个较长的单词，我们称这个单词为继承词（successor）。比如，词根 dis 跟随着单词 able（能够），可以形成新的单词 disable（不能够）。

这个游戏的玩法是这样的，给定一个由许多词根组成的字典和一个句子。你需要将句子中的所有继承词用词根替换掉。如果继承词中有许多形成它的词根，则用最短的词根替换它。

例如，字典为["cat", "bat", "rat"]，句子为"the cattle was rattled by the battery"，经过替换，输出句子为"the cat was rat by the bat"。



小朗朗准备写一个程序来解决这个问题，你可以帮帮他吗？

3.5.1 问题求解

先来看一下暴力破解法，由于输入是一个句子，所以需要先把句子转换为列表，代码如下：

```
01 def replaceWords(dict, sentence):
02       """
03       :type dict: List[str]
04       :type sentence: str
05       :rtype: str
06       """
07       s = sentence.split(" ")
```

对于字典中的每一个词根来说，依次判断句子中的每个单词是否以这个词根开头，如果以它开头的话，就把该单词用词根替换掉。假如该词根的位数为 n，可以通过截取第 i 个单词（s[i]）的前 n 位和该词根进行比较以判断是否相等，代码如下：

```
01 item=#某一个词根
02 for i in range(len(s)):
03   n = len(item)
04   if item == s[i][:n]:
05       s[i] = item
```

再用一个循环遍历所有的词根，对于每个词根都执行相同的操作，最后把所有的代码拼接起来即可，拼接字符串可以使用字符串的 join 函数来完成，完整代码如下所示：

```
01 def replaceWords(dict, sentence):
02       """
03       :type dict: List[str]
04       :type sentence: str
05       :rtype: str
06       """
07       s = sentence.split(" ")
08       for item in dict:
09          for i in range(len(s)):
10             n = len(item)
11             if item == s[i][:n]:
12                s[i] = item
13       return " ".join(s)
```

上面的代码虽然能够解决问题，但是效率太低。对于大数据量的问题，肯定是无法完成任务的，那么该如何优化算法呢？

该问题的本质还是一个查找问题，那么来看一下哈希能否对刚才的查找过程进行优化。

在暴力破解法中，对于每一个词根，需要在所有的单词中截取相应位数的子字符串，假如词根数量很多或者单词数量很多，双重循环的效率会比较低。在这个过程中，可以把用单词去匹配词根的过程通过哈希来优化。

首先建立两个字典，并做好初始化工作。在建立字典时我们使用 collections.defaultdict()建立一个默认字典，它本身是一个字典，只不过 Python 会自动为它的键赋一个初始值，也就是说，你不显式地为字典的键赋初值 Python 不会报错，代码如下：

```
01 def replaceWords(dict, sentence):
02       """
03       :type dict: List[str]
04       :type sentence: str
05       :rtype: str
06       """
07       d = collections.defaultdict(set)
08       s = collections.defaultdict(int)
09        sentence = sentence.split()
```

接下来，以每一个词根的首字母为键，把每一个词根放到该键所对应的值中去，这里的值是一个集合（set），同时记录下该首字母所对应的词根的最大长度是多少，这样一来，之后对比单词的前缀时，只需要看这个最大长度即可，代码如下：

```
01 for w in dict:
02    d[w[0]].add(w)
03    s[w[0]] = max(s[w[0]], len(w))
```

两个字典建立好了，如何使用它们呢？首先把每个单词拿出来，查找以该单词开头的词根是否能够和这个单词匹配，s 字典里记录了以某个字母开头的词根的最大长度，从第一位开始截取一直截取到最大长度，如果发现这个子字符串确实是一个词根的话，则修改单词，并结束该轮循环。为了方便记录下标和读取单词，使用 enumerate()函数来遍历句子，它会把索引放到第一个变量，把元素放到第二个变量，代码如下：

```
for i, w in enumerate(sentence):
          for j in range(s[w[0]]):
              if w[:j+1] in d[w[0]]:
                  sentence[i] = w[:j+1]
break
```

最后通过字符串的 join 方法连接各个单词即可。

```
return ' '.join(sentence)
```

3.5.2 最终代码

这个游戏的最终代码如代码 3.5 所示。

代码 3.5 神奇的词根

```
def replaceWords(dict, sentence):
    """
    :type dict: List[str]
    :type sentence: str
    :rtype: str
    """
    d = collections.defaultdict(set)
    s = collections.defaultdict(int)
    sentence = sentence.split()
    for w in dict:
        print(w[0])
        d[w[0]].add(w)
        s[w[0]] = max(s[w[0]], len(w))
    for i, w in enumerate(sentence):
        for j in range(s[w[0]]):
            if w[:j+1] in d[w[0]]:
                sentence[i] = w[:j+1]
                break
    return ' '.join(sentence)
```

第 4 章

深度优先遍历算法

深度优先遍历算法是经典的图论算法，深度优先遍历算法的搜索逻辑和它的名字一样，只要有可能，就尽量深入搜索，直到找到答案，或者尝试了所有可能后确定没有解。

本章主要涉及的问题有如下几个。

- 什么是深度优先遍历：简单介绍什么是深度优先遍历算法。
- 二叉树：详细介绍什么是二叉树。
- 怎么抓住小偷：树的深度优先搜索问题。
- 二叉树中的最大路径和：树的深度优先搜索问题。
- 最大的岛屿：连通性问题。

4.1 什么是深度优先遍历

首先，我们来看一下什么是深度优先遍历。

小朗朗准备暑假出去玩，有了几个备选目的地，但还没最终想好去哪里。小朗朗开始认真地做起旅游攻略，他的备选目的地有三亚、马尔代夫、巴厘岛、塞班岛，由于预算有限，他先查询去三亚的机票价格，如图 4.1 所示。

查好了机票价格，接下来，查询三亚的酒店价格，如图 4.2 所示。

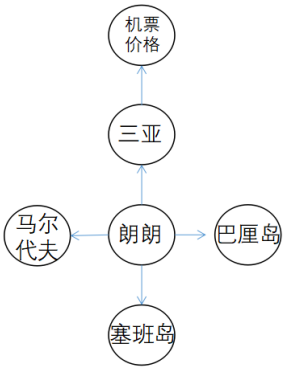


图4.1　旅游攻略1

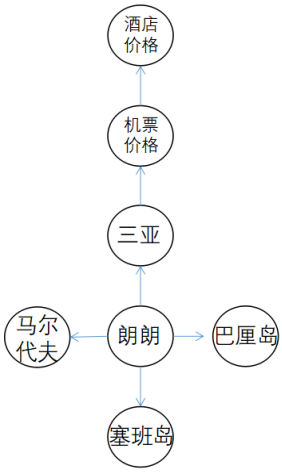


图4.2　旅游攻略2

查完了三亚的价格，再去查询马尔代夫的机票价格和酒店价格，如图 4.3 所示。

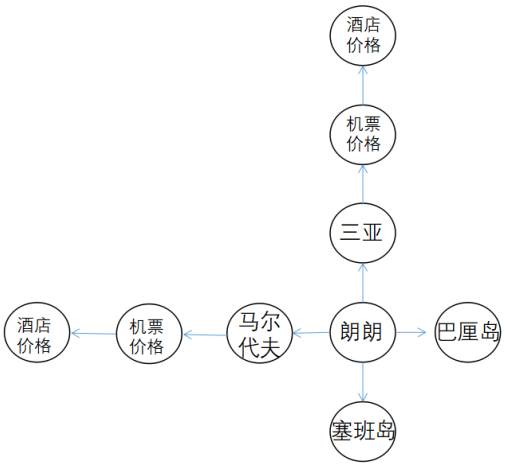


图4.3　旅游攻略3

小朗朗在做旅游规划时所使用的方法其实就是我们所说的深度优先遍历算法，沿着一个目的地不断地检索，直到把所有问题都解决，再去检索下一个目的地。

搜索问题的本质就是试探问题的所有可能选择，按照特定的规律和顺序，不断地去搜索答案，直到找到问题的解。如果把所有可能都走了一遍，也没有找到解，就说明这个问题没有解。

注意：深度优先遍历问题一定要按照规则尝试所有的可能才行。

深度优先遍历算法是经典的图论算法，从某个节点 v 出发开始进行搜索，不断搜索直到该节点的所有边都被遍历完。当节点 v 的所有边都被遍历以后，深度优先遍历算法则需要回溯到 v 的前驱节点，来继续搜索这个前驱节点的其他边。

如果还存在尚未被遍历的节点，则深度优先遍历算法将会按照统一的规则从这些剩下的节点中选择一个节点再重复同样的遍历过程。这样的搜索过程从最开始的节点一直持续到最后一个节点的所有边都遍历完。

接下来，我们来看几个深度优先搜索算法的问题。

4.2　二叉树

二叉树是一种特殊的数据结构。常见的数据结构包含数组、链表、图、队列、散列表与树。二叉树属于树结构。二叉树中的每一个节点都有两个分支，称为“左子树”与“右子树”。如图 4.4 所示，二叉树中的每一层最多有 2^n-1 个节点。和普通树不同，普通树的节点没有分支限制，并且普通树的节点没有左、右子树之分。

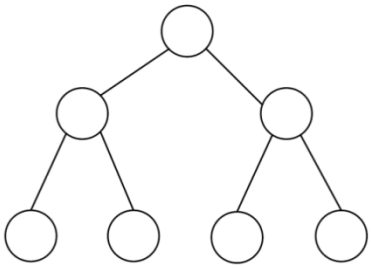


图4.4　二叉树

4.2.1　二叉树的类型

二叉树有几个特殊的类型：空二叉树、满二叉树、完全二叉树、完美二叉树、平衡二叉树。

空二叉树有零个节点。图 4.4 所示的是一棵完美二叉树（Perfect Binary Tree），每一层的节点都是满的。图 4.5 所示的是一棵满二叉树（Full Binary Tree），每一个节点都有零或两个子节点，也就是说，没有只有一个子节点的节点，国内对满二叉树和完美二叉树的定义是一样的，本文按照国外定义。

图 4.6 所示的是一棵完全二叉树（Complete Binary Tree），除了最后一层，每一层的节点都是满的，并且最后一层的节点全部从左排序。

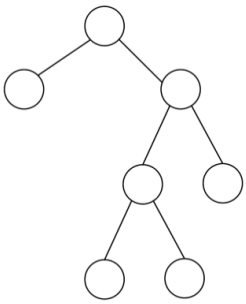


图4.5　满二叉树

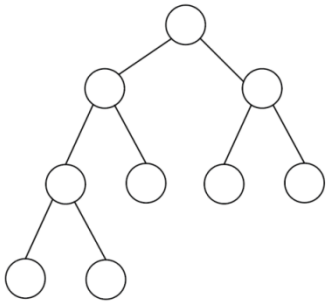


图4.6　完全二叉树

图 4.7 所示的既不是满二叉树也不是完全二叉树。不过，它是一棵平衡二叉树，每个节点的两个子树的深度相差不超过 1。

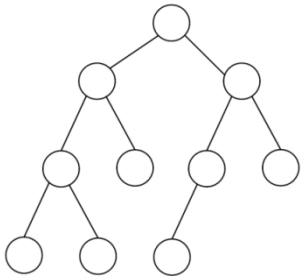


图4.7　平衡二叉树

4.2.2　二叉树的相关术语

二叉树的相关术语如表 4.1 所示。

表 4.1　二叉树的相关术语

术语	解释
度	节点的度为节点的子树个数
叶子节点	度为零的节点
分支节点	度不为零的节点
孩子节点	节点下的两个子节点
双亲节点	节点上一层的源头节点
兄弟节点	拥有同一个双亲节点的节点
根	二叉树的源头节点
深度	二叉树中节点的层的数量

4.2.3　二叉树的节点代码

因为每一个节点都与两个子节点相连，所以我们只需要拥有根节点就能找到二叉树上的任意节点。每一个节点的定义都是相同的，代码如下：

```
01 class Node:                                    #二叉树节点的定义
02    def   init  (self, x):
03        self.val = x                            #节点值
04        self.left = None                        #左侧子节点
05        self.right = None                       #右侧子节点
```

4.2.4　二叉树的遍历顺序

二叉树有三种遍历顺序：DLR（先序）、LDR（中序）、LRD（后序）。遍历的意思是不重复地走遍二叉树的所有节点。L、R、D 分别代表左子树、右子树和根。先、中、后代表的是根节点的位置。DLR 代表先遍历根，再遍历左子树，最后遍历右子树。让我们以 DLR 的顺序遍历图 4.8，节点的访问顺序为：1-2-4-5-3-6-7。

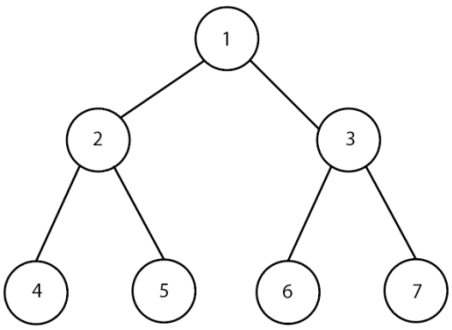


图4.8　二叉树

LDR 代表先遍历左子树，再遍历根，最后遍历右子树。以 LDR 的顺序遍历图 4.8，节点的访问顺序为：4-2-5-1-6-3-7。LRD 代表先遍历左子树，再遍历右子树，最后遍历根。以 LRD 的顺序遍历图 4.8，节点的访问顺序为：4-5-2-6-7-3-1。

4.2.5　深度优先遍历与广度优先遍历

我们之前讲到的前序、中序和后序遍历都属于深度优先遍历。在深度优先遍历中，我们从根节点出发直奔最远的节点。在广度优先遍历中，我们首先访问离根节点最近的节点，按层递进。还是以图 4.8 为例，让我们以广度优先的顺序遍历该图，顺序为：1-2-3-4-5-6-7。

4.3 怎么抓住小偷

有一个绝顶狡猾的小偷瞄准了一个高档小区。如图 4.9 所示，这个小区的别墅以二叉树的结构坐落。除了第一栋别墅，每一栋别墅都与另一栋“源头”别墅连接。一旦小偷闯入两栋相接的别墅，警铃就会被触发。圆圈里的数字代表财产，狡猾的小偷已经计划好了路线，准备在不触发警报的前提下偷最多的钱。别墅小区的保安部门最近得到了风声，说有一个世纪大盗瞄上了别墅区。据说此小偷神无踪影，武功高强，所以保安打算在小偷的必经之路上布置警卫，不动声色地抓住小偷。问题是，小偷会怎么选路线呢？

如图 4.10 所示，小偷肯定会选择灰色的这两栋别墅，偷到 4+5=9 万元。在这一节中，我们写一个程序来输出小偷经过路线上的金额总和。

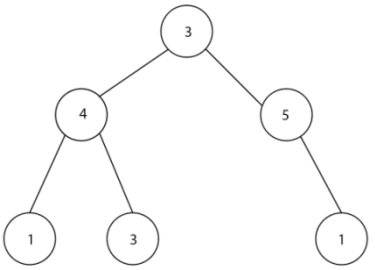


图4.9　小区模型

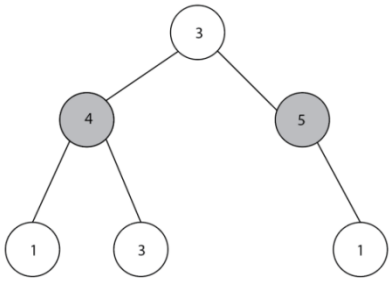


图4.10　闯入的房子

再来看一个例子，如果别墅布置如图 4.11 所示，那么小偷肯定会闯入以下三栋别墅，得到 3+3+1=7 万元。

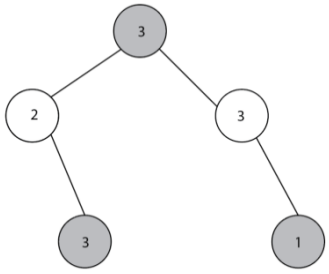


图4.11　例子二

4.3.1 解题思路

首先，我们分析一下前提条件。前面说到两栋相连的别墅被盗时警铃会被触发。如图 4.12 所示，把二叉树分层。

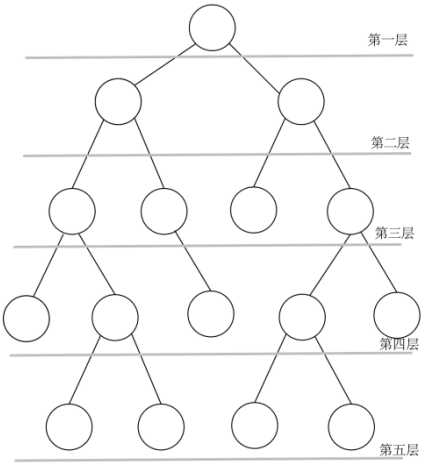


图4.12　分层

如果小偷闯入第一层，那么他就不能偷第二层的房子。他可以选择偷第三层或第四层的房子。小偷需要统观全局，因为他在每一层的决定都影响下一层。如图 4.13 所示，小偷有不同的选择方案。

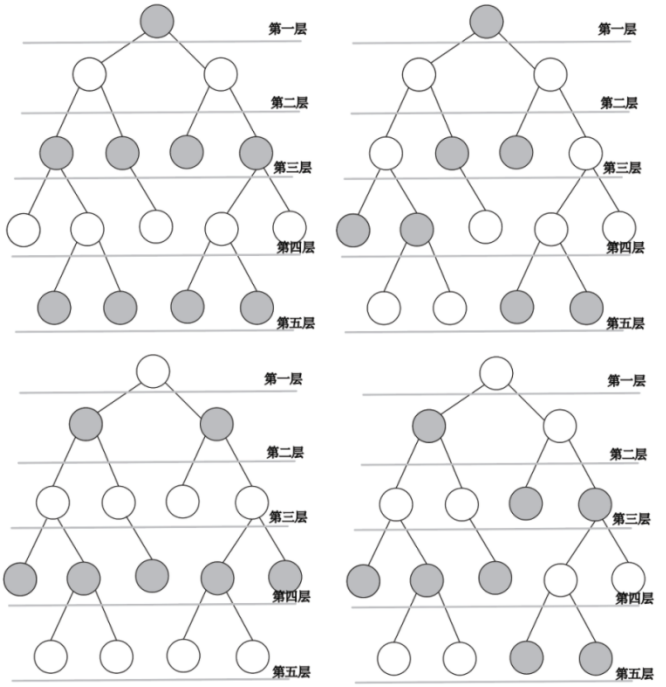


图4.13　选择方案

在每一个节点，小偷都需要做一个选择：偷还是不偷。小偷需要比较如果偷能得到多少钱和不偷能得到多少钱。如果偷了这个节点，这个节点的子节点就不能再偷了。

让我们给每一个节点两个值：偷得到的钱和不偷得到的钱。如图 4.14 所示，例子中的每一个节点旁边都有偷：0 与不偷：0 两个值。让我们从底层开始，更改节点 4、5 和 6 的值。如果偷，得到相对应的钱；如果不偷，则得到 0。

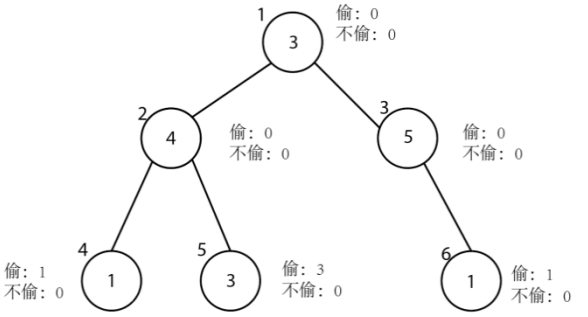


图4.14　解题思路1

如图 4.15 所示，上移一层，设节点 2 与节点 3 的值。在节点 2，如果偷，小偷将会得到 4 万元，但与节点 4 与节点 5 的钱无缘；如果不偷，小偷可以偷节点 4 与节点 5 的钱，得到 4 万元。再看节点 3，如果偷，小偷得到 5 万元；如果不偷，小偷得到节点 6 的 1 万元。

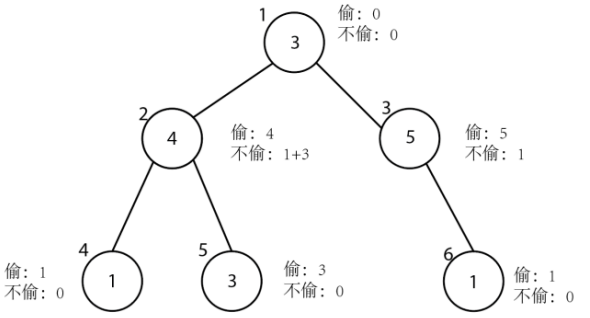


图4.15　解题思路2

如图 4.16 所示，设根节点 1 的值。如果偷，将会得到节点 1 的 3 万元、节点 4 与节点 5 的 4 万元，以及节点 6 的 1 万元；如果不偷，可以得到节点 2 的 4 万元，以及节点 3 的 5 万元。权衡金额，小偷应该会采取图 4.17 所示的两个措施之一。

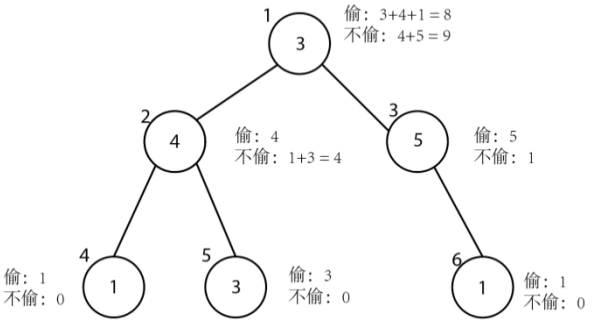


图4.16　解题思路3

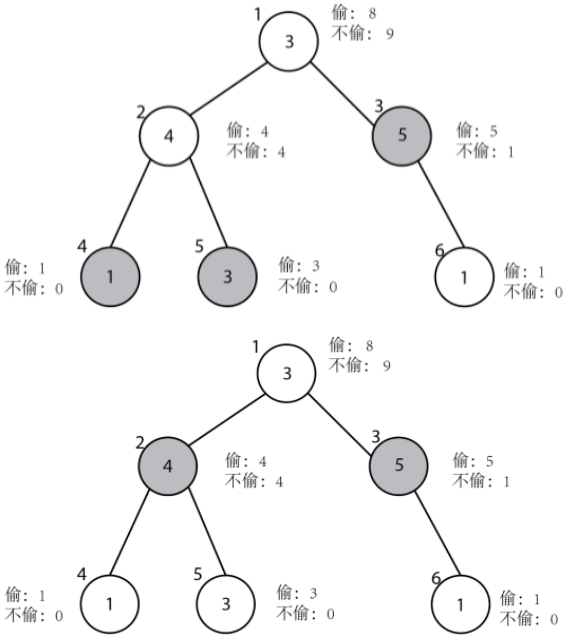


图4.17　解题思路4

在以上的分析中我们发现了一个规律。每一个节点的偷值都是：左侧子节点的不偷值+右侧子节点的不偷值+节点的财富；每一个节点的不偷值都是：左侧子节点的最大值+右侧子节点的最大值。比如，节点 1 的偷值是 4（节点 2 的不偷值）+1（节点 3 的不偷值）+3（节点 1 的财富）。节点 1 的不偷值是 4（节点 2 的最大值）+5（节点 3 的最大值）。

如果节点的定义为：

```
class TreeNode:                                    #二叉树节点的定义
  def   init  (self, x):
      self.val = x                                 #财富值
      self.left = None                             #左侧子节点
      self.right = None                            #右侧子节点
```

那么任何一个节点（用 root 表示）的偷值（robValue）与不偷值（skipValue）则是：

```
robValue = root.val + root.left.skipValue +  root.right.skipValue
skipValue=max(root.left.robValue,root.left.skipValue)+
max(root.right.robValue,root.right.skipValue)
```

现在需要解决的问题是：如何得到子节点的偷值与不偷值，也就是说，如何得到 root.left.robValue、root.left.skipValue、root.right.robValue 与 root.right.skipValue。

答案很简单，找到子节点的子节点的偷值与不偷值就可以了，也就是说，root.left.left.robValue、root.left.left.skipValue、root.left.right.robValue、root.left.right.skipValue、root.right.left.robValue…… 即利用递归来深度优先遍历二叉树。

4.3.2 从思路到代码

现在我们尝试把之前的思路写出来，如代码 4.1 所示。

代码 4.1 二叉树别墅小区的小偷

```
 def rob(self, root):
    a = self.helper(root) #a 是一个二维数组，为 root 的[偷值，不偷值]
    return max(a[0],a[1]) #返回两个值的最大值，此值为小偷最终获得的财富值。
 def helper(self,root): #参数为 root 节点，helper 方法输出一个二维数组：root 节点的[偷值，不偷值]
    if(root==None):        #如果 root 节点为空，输出[0,0]
       return [0,0]
    left = self.helper(root.left)  #left 是一个二维数组，为 root 左侧子节点的[偷值，不偷值]
    right = self.helper(root.right) #right 也是一个二维数组，为 root 右侧子节点的[偷值，不偷值]
    robValue = root.val + left[1]+ right[1]      #root 的偷值
    skipValue = max(left[0],left[1]) + max(right[0],right[1])     #root 的不偷值
    return [robValue, skipValue]                 #输出小偷可获得的最大金额
```

4.4 二叉树中的最大路径和

小朗朗在打怪兽的路上，怪兽们的位置以二叉树的形式展示。小朗朗最少打一只怪兽，并且不能重复打已经交过手的怪兽。每打败一只怪兽都有对应的奖励，但如果失败也有惩罚，小朗朗的目的是尽可能多地得到奖励，或者最少的惩罚。这一节的目的是写一个程序，输出最大

可能的路径和。

如果你是小朗朗，你会选择哪一条打怪兽的路呢？如图 4.18 所示，按照图中的安排，小朗朗应该选择灰色这条路，输出 7。如图 4.19 所示，在这个特定的例子中，小朗朗的最佳路线不经过根节点，输出 24。如图 4.20 所示，在这个例子中，小朗朗的最佳路线只是一个节点，输出 6。

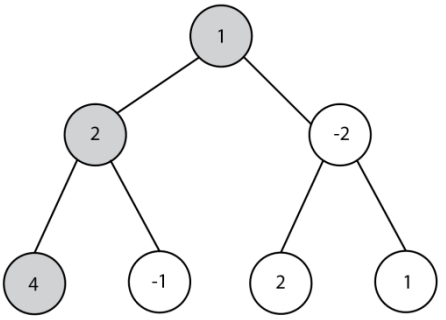


图4.18　例子1

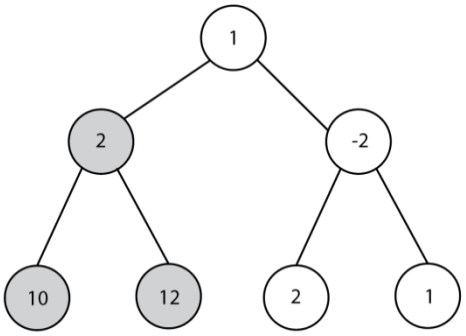


图4.19　例子2

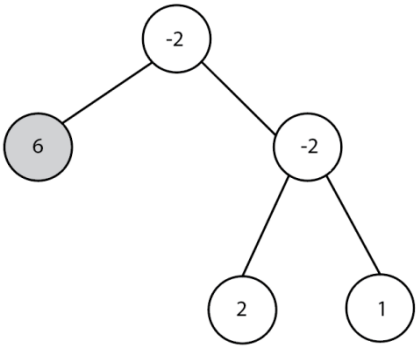


图4.20　例子3

4.4.1　解题思路

让我们来分析一下题目。节点的值可负可正，我们要保证路线总和最大。走过的节点不能再次访问，也就是说，如果我们访问了一个叶子节点（没有子节点的节点），路线就没有下一步了。当然，我们不一定在叶子节点终止路线，随时都可以停止，及时止损。

在每一个节点上都有两种可能：继续或不继续。可以继续的条件为线路的两端至少有一端不为叶子节点，不继续没有条件约束。

让我们来看一个例子。每一个节点都有两个值：停或不停。“停”为不继续的路线最大值，

“不停”为可延伸路线的最大值。如图 4.21 所示，在底层（节点 4、5、6、7），节点的停与不停值都是一样的。

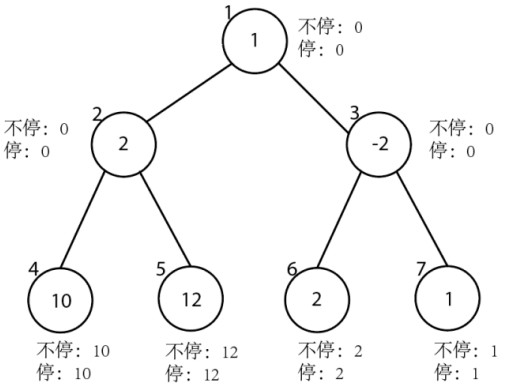


图4.21　底层节点

我们上移一层，到节点 2、3。先来看节点 2，如图 4.22 所示，停与不停各包含三个值。这是因为停与不停各有三条不同的路线。最后我们会各选择三者中的最大值，不过现在先来看一下为什么是这些值。

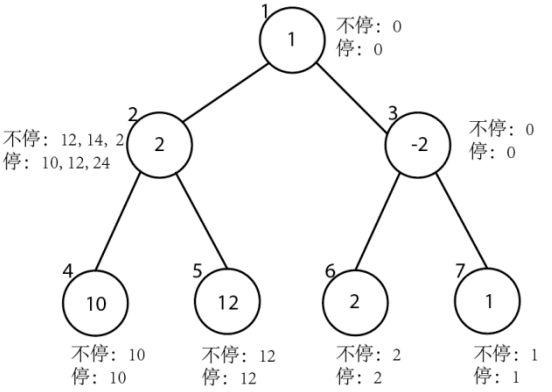


图4.22　节点2处的情况

首先，节点 2 的停值。如图 4.23 所示，不继续路线有三种情况：在节点 4 停止、在节点 5 停止、经过节点 4 与节点 2 后在节点 5 停止。因此，停有三个可能值：10、12、24。

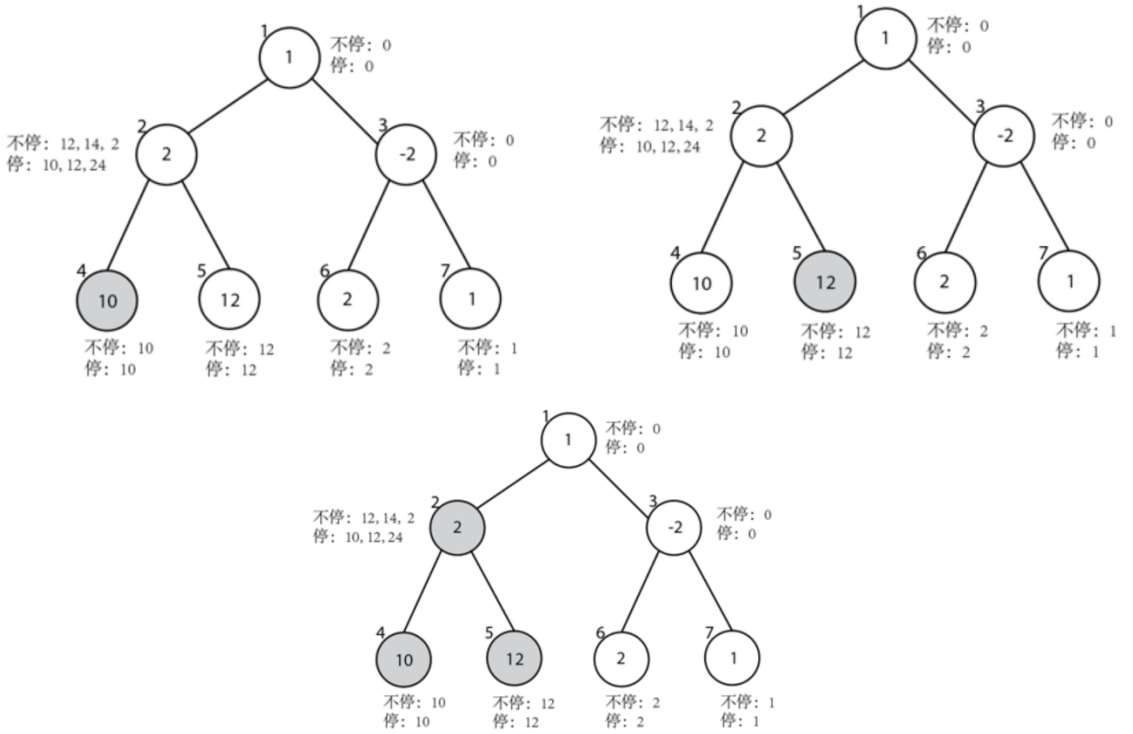


图4.23　不继续的三种路线

再来看节点 2 的不停值。如图 4.24 所示，可延伸路线也有三种情况：经过节点 2 与 4、经过节点 2 与 5、经过节点 2。因此，不停有三个可能值：12，14，2。

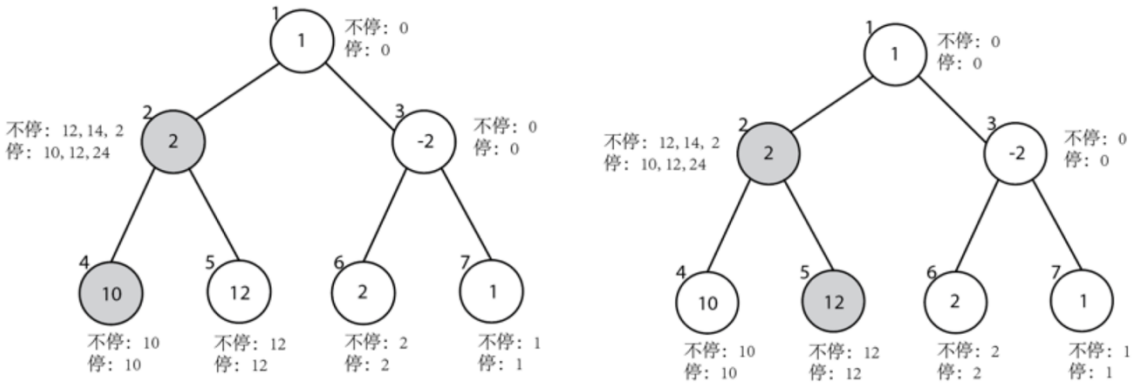


图4.24　可延伸的三种路线

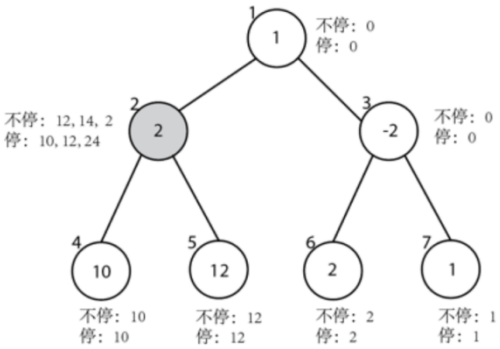


图4.24　可延伸的三种路线（续）

选择节点 2 停值与不停值的最大值：14 与 24。如图 4.25 所示，来看同一层的节点 3，做同样的步骤。

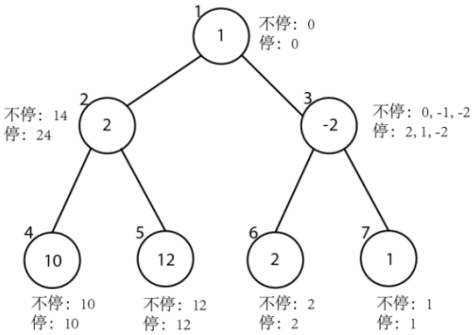


图4.25　节点3处的情况

先来看节点 3 的停值。如图 4.26 所示，不继续路线有这三种情况：在节点 6 停止、在节点 7 停止、经过节点 6 与节点 3 后在节点 7 停止。因此，停有三个可能值：2、1、−2。

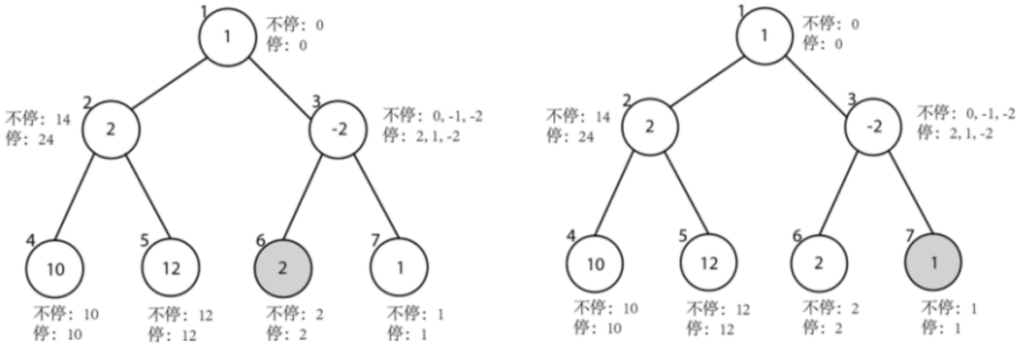


图4.26　节点3不继续的三种路线

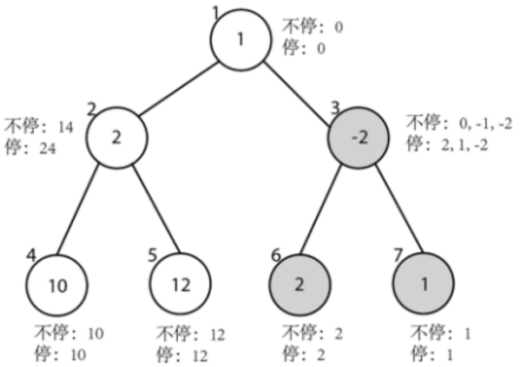


图4.26　节点3不继续的三种路线（续）

再来看节点 3 的不停值。如图 4.27 所示，节点 3 的可延伸路线也有三种情况：经过节点 3，经过节点 3 与 6、经过节点 3 与 7。因此，不停有三个可能值：−2、0、−1。

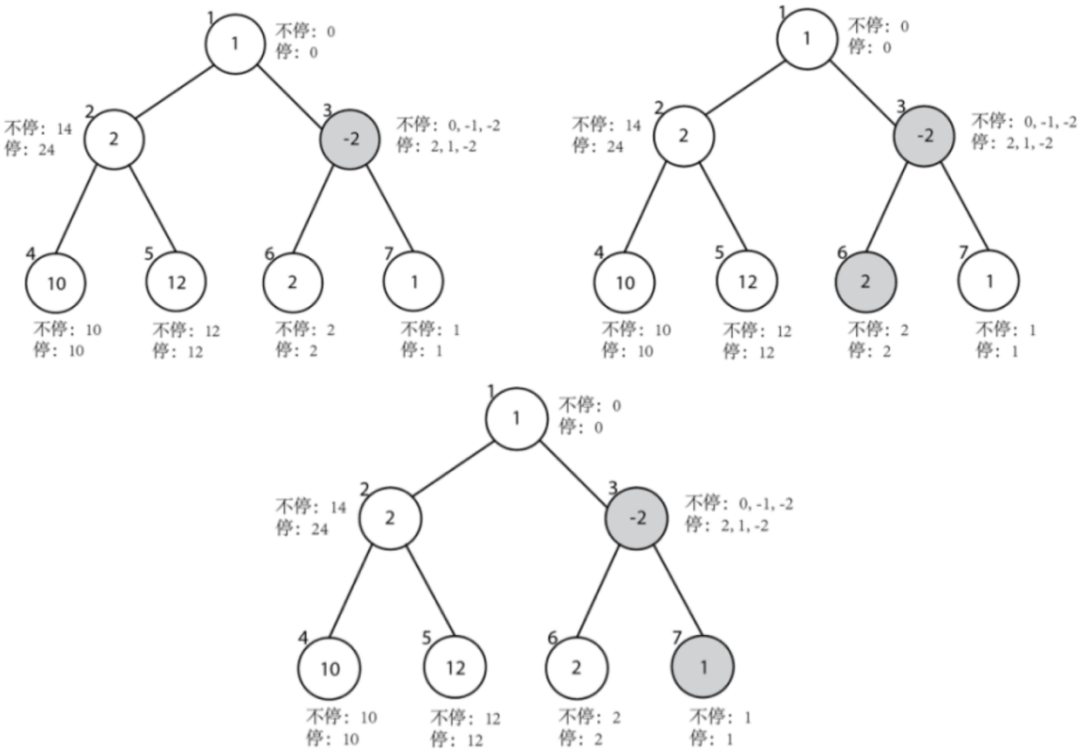


图4.27　节点3可延伸的三种路线

最后，到达根节点。根节点的可延续路线有三种可能（但两个值重复），不延续路线有五种可能。我们分别来分析一下，如图 4.28 所示。

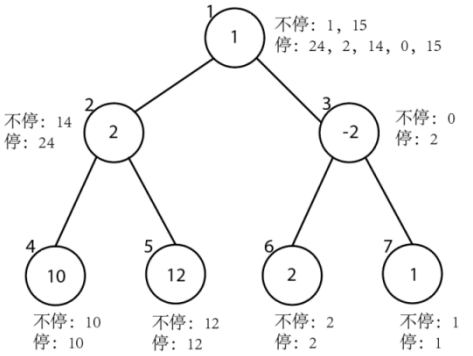


图4.28　根节点处的情况

首先，节点 1 的不停值。如图 4.29 所示，可延伸路线有三种情况：节点 1、节点 1 加上左子节点的不停路线、节点 1 加上右子节点的不停路线。因此，不停有两个可能值：1，15。

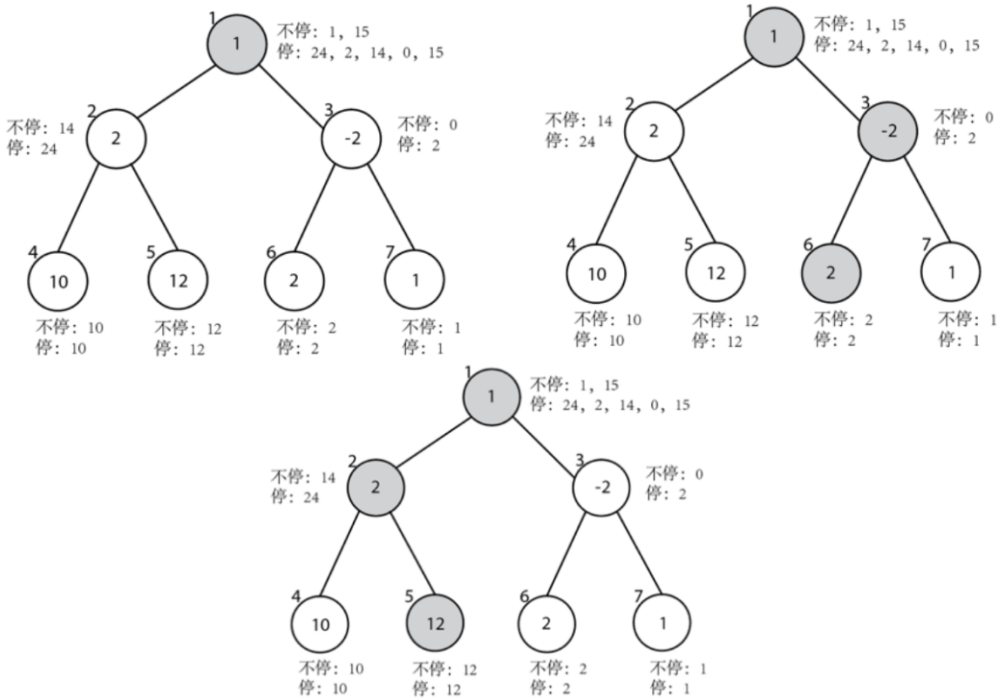


图4.29　根节点的可延伸路线

再来看节点 1 的停值。如图 4.30 所示，不延伸路线有五种情况：左子节点的不继续路线、右子节点的不继续路线、左子节点的继续路线、右子节点的继续路线、根节点加上两个子节点的继续路线。因此，节点 1 的停值有五个可能值：24、2、14、0、15。

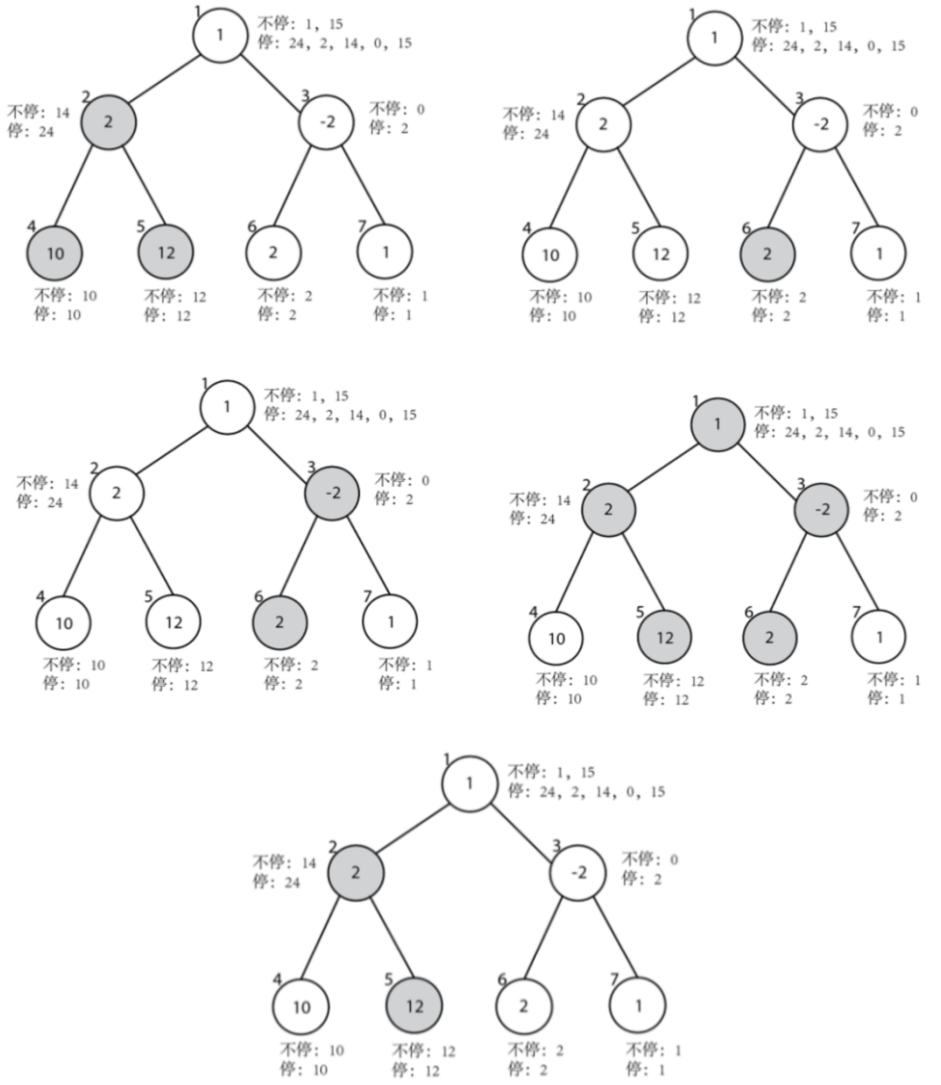


图4.30　根节点的不继续路线

遍历完所有的节点，我们输出根节点的最大值（不停值与停值的最大值），也就是 24，相对应的路线是节点 4-节点 2-节点 5，如图 4.31 所示。

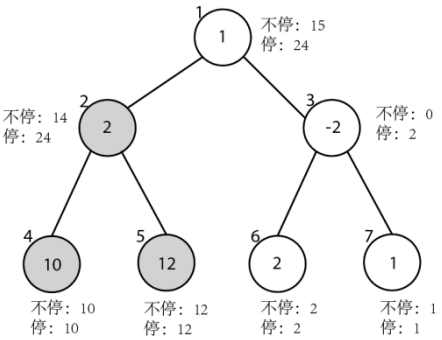


图4.31　输出最大值

再来看一个例子。如图 4.32 所示，每一个节点还是标记两个值：停值和不停值。

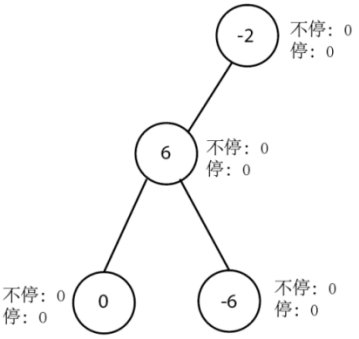


图4.32　例子2

首先，底层的叶节点的停值与不停值都为节点值：0 与−6。第二层节点的可延续路线与停止路线各有三种可能。图 4.33 展示了可延续路线的三种情况，对应的不停值为：6、0、6。图 4.34 展示了不继续路线的三种情况，对应的停值为：0、−6、0。

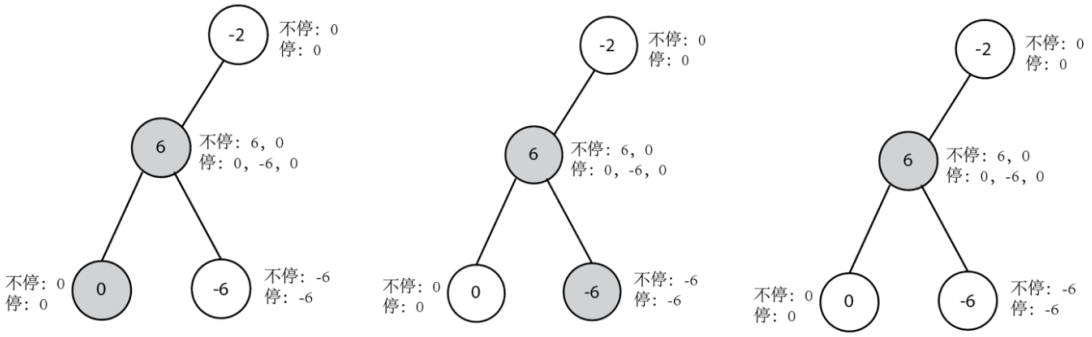


图4.33　不停值

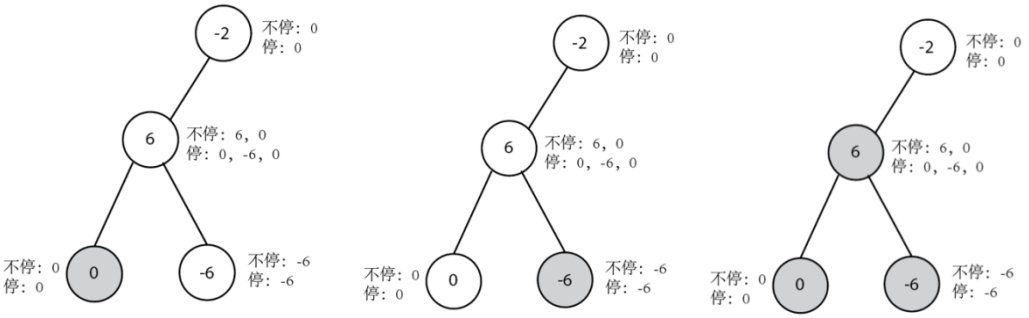


图4.34　停值

最后，让我们来看一下根节点。根节点有两个可能的停值，三个可能的不停值。如图 4.35 所示，根节点的不继续路线有两种可能：左子节点的继续路线、左子节点的不继续路线。

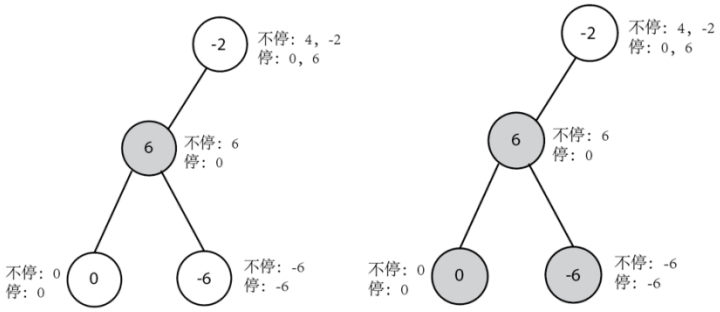


图4.35　根节点的停值

在上一个例子中，不继续路线还包括根节点加两个子节点的继续路线，但是因为这个例子的根节点没有右子节点，所以不继续路线只有两种情况。因此，节点 1 的停值有两个可能值：6、0。

如图 4.36 所示，根节点的可延伸路线有两种可能：根节点加上左子节点的不继续路线、根节点。因此，节点 1 的不停值有两个可能值：4、−2。

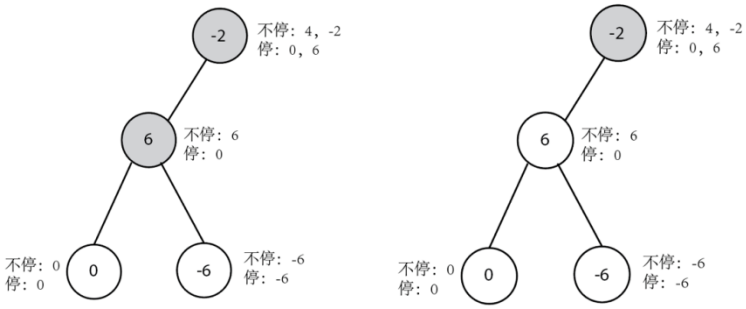


图4.36　根节点的不停值

最后，我们选择输出根节点的最大值，6。如图 4.37 所示，这个例子的最佳路线是一个单个节点。

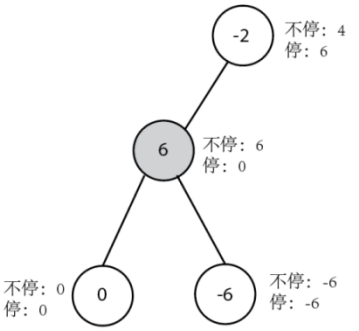


图4.37　例子2的输出

我们通过深度优先算法遍历了所有的节点，深度优先算法的想法是首先遍历最低层的节点，然后逐步上移到根节点。

每遍历一个节点，输出两个值以便之后的节点做决定，一个是停值，一个是不停值。我们现在的工作是找到一个适用于所有节点的停值与不停值的输出方法。来看以下代码：

```
01  class TreeNode:                          #二叉树节点定义
02    def   init  (self, x):
03        self.val = x                       #节点值
04        self.left = None                   #左子节点
05        self.right = None                  #右子节点
06  def helper(self, root):                  #helper 方法，输出一个二维数组 [ 不停值，停值 ]
07   leftY,leftN = self.helper(root.left)    #得到 root 左子节点的不停值与停值
08       rightY,rightN = self.helper(root.right)      #得到 root 右子节点的不停值与停值
09       yes = max(root.val+leftY,root.val+rightY,root.val)          #不停值（yes 代表继续）
10       no = max(leftN,rightN,leftY,rightY,root.val+leftY+rightY) #停值（no 代表不继续）
11       return yes,no                       #输出 root 节点的[ 不停值，停值 ]
```

root 节点的不停值是以下三个值中的最大值：root 值、root 值+左子节点的不停值、root 值+右子节点的不停值。root 节点的停值是以下五个值中的最大值：左子节点的不停值、右子节点的不停值、左子节点的停值、右子节点的停值、root 值+左右子节点的不停值。

4.4.2　完整代码

到现在为止，我们还没有考虑空节点的情况。如果节点为 None，我们应该输出两个最小值，让 max()方法过滤掉这个空节点。完整代码如代码 4.2 所示。

代码 4.2　二叉树中的最大路径和

```
01 class Solution:
02   def maxPathSum(self, root):        #输出最大路径和的方法
03       return max(self.helper(root))  #调用 helper 方法，传入根节点，输出返回的两个值中的最大值
04   def helper(self, root):            #helper 方法，输出一个二维数组 [ 不停值，停值 ]
05       if root==None:                 #如果节点为空，输出两个最小值
06           return float('-int'), float('-int')
08       leftY,leftN = self.helper(root.left)         #得到左子节点的不停值与停值
10       rightY,rightN = self.helper(root.right)      #得到右子节点的不停值与停值
11       yes = max(root.val+leftY,root.val+rightY,root.val)            #不停值
12       no = max(leftN,rightN,leftY,rightY,root.val+leftY+rightY)     #停值
13       return yes,no                  #输出 [ 不停值，停值 ]
```

4.5　最大的岛屿

经过深度优先遍历算法，小朗朗最终选择去马尔代夫，马尔代夫有上百个岛，小朗朗想知道哪个岛是最大的岛屿。这次又需要深度优先遍历算法来帮忙了。如图 4.38 所示，为了方便计算，我们使用一个二维数组来表示马尔代夫的一片海域，使用 0 表示水面，1 表示陆地，目前这块区域一共有面积各不相同的 5 个小岛，我们的任务是找到其中最大的岛屿。

注意：岛屿是指上下左右四个方向相连接的陆地。

那么对于这个例子来说，左上角的那个岛屿就是最大的岛屿，如图 4.39 所示。

0	0	0	0	1	1	0
0	1	1	0	1	1	0
0	1	1	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1

图4.38　岛屿示意图

0	0	0	0	1	1	0
0	1	1	0	1	1	0
0	1	1	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1

图4.39　最大的岛屿示意图

4.5.1　解题思路

首先，我们分析一下这个问题。问题是找到最大的岛屿，那么一定要把每个岛屿的面积算出来，然后在其中找出最大的即可。

假如让人类去干这件事，我们该如何干呢？我们要逐个登上每个岛屿，然后开始测量面积，最后从中选择出一个最大的岛屿。听起来好像是废话，但是写程序其实就是把人类的做法用编程语言写出来，变为计算机的解题思路。

对于人类来说，一眼望过去是陆地，那么我们就登陆岛屿开始测量。但是对于计算机来说怎么发现岛屿呢，人类一眼望过去的过程，对于计算机来说只能靠遍历来扫描了。我们采用从左向右，从上到下的扫描方式查找岛屿的位置。那么，一旦发现 1 就说明发现了岛屿，那么就登陆岛屿开始测量，如图 4.40 所示。

图4.40　岛屿搜索

同时，还需要一个变量来记录下最大岛屿的面积，然后在程序的最后把它返回，那么来看一下查找岛屿的代码吧：

```
01 def maxAreaOfIsland(self, grid):
02   self.maxArea= 0              #存储最大岛屿的面积
03   row = len(grid)              #存储地图的行数
04   col = len(grid[0])           #存储地图的列数
05   for i in range(row):
06        for j in range(col):  #开始从左到右、从上到下地搜索岛屿
07             if grid[i][j] == 1:    #如果发现了陆地
08                                    #测量岛屿面积，核心代码
09   return self.maxArea          #最后返回最大岛屿的面积
```

登陆以后该如何测量呢，由于岛屿是由上下左右四个方向的陆地组成的，而不是由斜着的陆地组成，那么我们可以规定一种查找顺序，全部按照上下左右四个方向的顺序查找。

现在就可以开始用深度优先遍历算法测算岛屿的面积了。从登陆岛屿的第一个平方米开始我们就要按照上面规定的顺序不断地计算岛屿的平方米数。计算完脚下的这一平方米，就要走到下一平方米，对于下一平方米的选择按照上下左右的顺序，直到走到不能走为止才换方向。为了不重复计算，每计算一平方米就把陆地改为 2，标记为已经访问过。

来看一个例子，如图 4.41 所示，最先找到的陆地是边框为粗体的地方。

接下来按理说我们需要去它的上面查找，但是向上走就超出地图的边界了。那么，我们就换一个方向，试试向下如何，发现向下没有越界，并且还是陆地，这时我们移动到下面的点，如图 4.42 所示。

0	0	0	0	**1**	1	0
0	1	1	0	1	1	0
0	1	1	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1

图4.41　找到第一片陆地

0	0	0	0	2	1	0
0	1	1	0	1	1	0
0	1	1	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1

图4.42　向下移动到第二片陆地

来到下面的陆地了，还要遵循一样的顺序，先尝试向上走，发现上面没有未访问的陆地，那么换一个方向再尝试向下走，发现下面不是陆地，再尝试向左走，也不是陆地，最后向右走发现是陆地，这时就可以移动到右面的陆地上，如图 4.43 所示。

再来看一下走到最后一片陆地时该如何做。到达最后一片陆地时，同样尝试向四个方向走去寻找陆地，但是环顾四周也没有发现新的陆地，这时该怎么办呢？回退到上一步的位置，再换一个方式走，如图 4.44 所示。同样的，上一步的位置也没有可以走的地方了，继续回退到最初点，整个搜索过程结束。这就是深度优先遍历的全部过程。

0	0	0	0	2	1	0
0	1	1	0	2	1	0
0	1	1	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1

图4.43　向右走移动到达第三片陆地

0	0	0	0	2	2	0
0	1	1	0	2	2	0
0	1	1	0	0	0	0
0	0	1	0	0	1	1
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	0	0	1	0	0	1

图4.44　无路可走则回退到上一步的位置

为此，我们重新定义一个新的函数来进行深度优先遍历算法，程序的第一步先标记此处为 2（已访问），然后开始四个方向的递归调用，注意递归的时候要避免越界和重复访问，来看一

下代码吧：

```
def dfs(self, k, z, current, grid):
    # k,z 为点的坐标
    # current 存储目前岛屿的面积，grid 为地图
   grid[k][z] = 2                                    #第一步先标记此处为已访问
   if k > 0 and grid[k-1][z] == 1:                   #向上走前先考查是否越界并且是否为陆地
       current=self.dfs(k-1, z, current+1, grid)  #递归调用函数，更新 x 左边和当前面积
   if k < (len(grid)-1) and grid[k+1][z] == 1:       #向下走前先考查是否越界并且是否为陆地
       current= self.dfs(k+1, z, current+1, grid)
   if z > 0 and grid[k][z - 1] == 1:                 #向左走前先考查是否越界并且是否为陆地
       current= self.dfs(k, z-1, current+1, grid)
   if z < (len(grid[0])-1) and grid[k][z+1] == 1: #向右走前先考查是否越界并且是否为陆地
       current= self.dfs(k, z+1, current+1, grid)
    self.maxArea= max(self.maxArea, current)         #更新最大面积变量
    return current
```

最后在主函数中调用这个深度优先遍历函数即可。

4.5.2 完整代码

综合以上分析，合并成代码 4.3 所示的完整代码。

代码 4.3 最大的岛屿

```
class Solution(object):
 def maxAreaOfIsland(self, grid):
     self.maxArea= 0             #存储最大岛屿的面积
     row = len(grid)             #存储地图的行数
     col = len(grid[0])          #存储地图的列数
     for i in range(row):
        for j in range(col):    #开始从左到右、从上到下地搜索岛屿
           if grid[i][j] == 1: #如果发现了陆地
               current = 1
               self.dfs(i, j, current, grid)   #测量岛屿面积，核心代码
     return self.maxArea         #最后返回最大岛屿的面积
 def dfs(self, k, z, current, grid):
 # k,z 为点的坐标
 # current 存储目前岛屿的面积，grid 为地图
     grid[k][z] = 2                            #第一步先标记此处为已访问
     if k > 0 and grid[k-1][z] == 1:           #向上走前先考查是否越界并且是否为陆地
           current= self.dfs(k-1, z, current+1, grid) #递归调用函数,更新 x 左边和当前面积
     if k < (len(grid)-1) and grid[k+1][z] == 1: #向下走前先考查是否越界并且是否为陆地
           current= self.dfs(k+1, z, current+1, grid)
```

```
20        if z > 0 and grid[k][z - 1] == 1:                #向左走前先考查是否越界并且是否为陆地
21              current= self.dfs(k, z-1, current+1, grid)
22        if z < (len(grid[0])-1) and grid[k][z+1] == 1:#向右走前先考查是否越界并且是否为陆地
23             current= self.dfs(k, z+1, current+1, grid)
24        self.maxArea= max(self.maxArea, current)       #更新最大面积变量
25        return current
```

深度优先算法的本质就是“一条路走到黑”，直到不能走才换另一条路，有点“不撞南墙不回头”的意思。

第 5 章

广度优先遍历算法

广度优先遍历与深度优先遍历类似，也是查询的方法之一，它也是从某个状态出发查询可以到达的所有状态。但不同于深度优先遍历，广度优先遍历总是先去查询距离初始状态最近的状态。

本章主要涉及的问题有如下几个。

- 选课的智慧：如何确定选课顺序。
- 寻找制高点：寻找制高点，抢占有利地形。
- 合法的括号：从非法括号序列中寻找合法括号序列。
- 树的右侧：从树的侧边观察树的伟岸身躯。

5.1 什么是广度优先遍历

首先，我们来看一下什么是广度优先遍历，和深度优先遍历算法类似，它也是一种搜索算法，但不同于深度优先遍历，广度优先遍历总是先去查询距离初始状态最近的状态。

还是以小朗朗去旅游为例，但是这次以广度优先遍历的方式来做旅游攻略。小朗朗的备选目的地有三亚、马尔代夫、巴厘岛、塞班岛，如图 5.1 所示。

由于预算有限，小朗朗先查询去各个地方的机票价格，如图 5.2 所示。

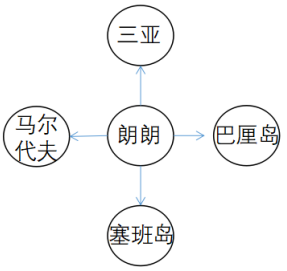


图5.1　旅游目的地

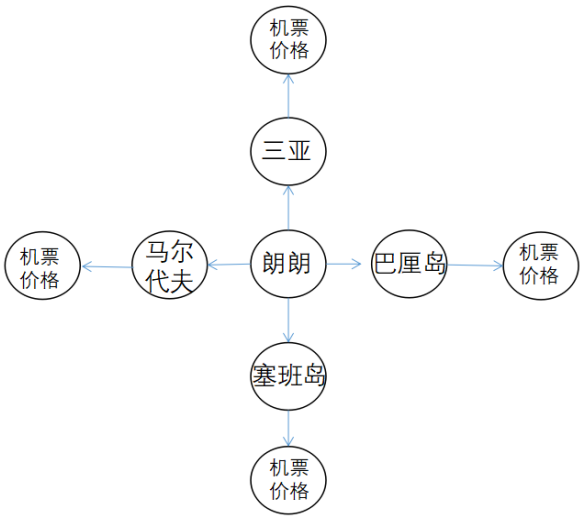


图5.2　先查机票价格

然后，小朗朗开始查询各个目的地的酒店价格，如图 5.3 所示。

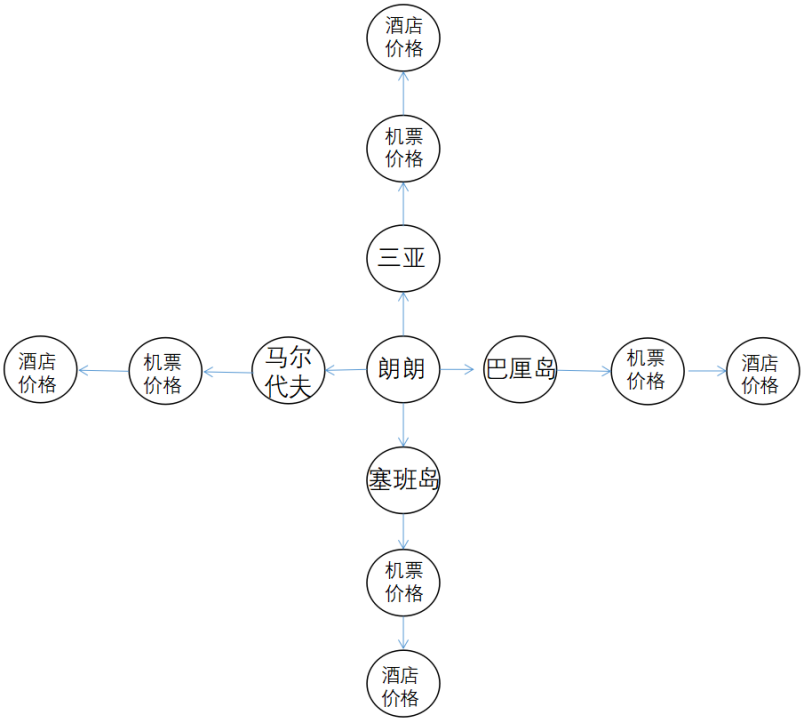


图5.3　再查酒店价格

对比第 4 章介绍的深度优先遍历算法，可以看出，广度优先遍历算法在搜索所有答案的时候是采用由近及远的方式。先访问离起始点最近的点，再访问远一些的点，就好像先访问走一步可以到达的点，再访问走两步可以到达的点。

因此，广度优先遍历算法也叫作层次遍历算法，一层一层地去寻找问题的答案。接下来，让我们来看几个广度优先遍历算法的例子吧。

5.2 选课的智慧

新学期伊始，小朗朗正在面临选课的问题，我们要学习计算机基础、数学、英语、算法、Java 等五门课程，其中学习算法前需要先学习 Java、英语，学 Java 前又需要学数学和计算机基础，如图 5.4 所示。那么小朗朗到底该如何选课呢？

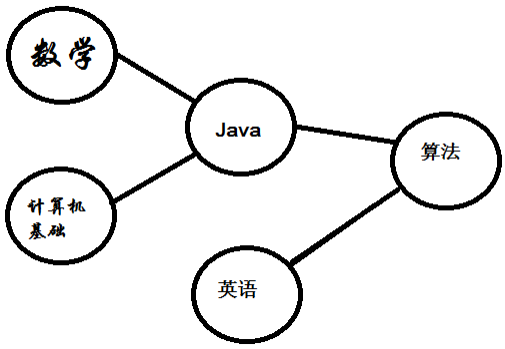


图5.4 选课图

我们先给每一门课设置一个编号，编号从 0 开始。先修课的关系如何表达呢，课程与课程间的依赖关系以一个二维数组的形式给出，如表 5.1 所示。例如数组中第 1 行第 3 列的 1 表示课程 0 是课程 2 的先修课。

表 5.1 课程之间的依赖关系

	0	1	2	3	4
0	0	0	**1**	0	0
1	0	0	1	0	0
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	0

5.2.1　广度优先遍历

解决这个问题之前，让我们先看一下什么是广度优先搜索。假如小朗朗遇到了一些法律相关的问题需要咨询，他就开始在自己的朋友圈中寻找律师朋友，小朗朗首先在自己的好朋友当中去寻找，如图 5.5 所示。

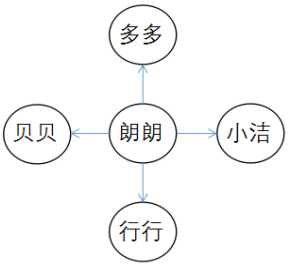


图5.5　朋友圈

为此小朗朗把他的好朋友加入考查列表，考查列表用我们最熟悉的数组来表示，如表 5.2 所示。

表 5.2　考查人员列表

多多	贝贝	行行	小洁		

注意：考查列表初始为空，依次把元素加入其中。

小朗朗依次查看他的每一个朋友，发现没有人是律师。小朗朗心想，自己的朋友中没有，或许朋友的朋友中有，接下来小朗朗又开始在他的朋友的朋友中搜索，如图 5.6 所示。

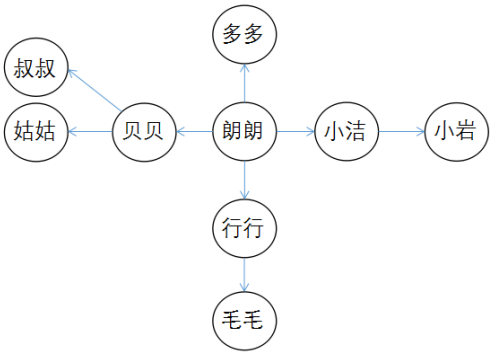


图5.6　扩展朋友圈

存放考查对象的数组变成下面的样子，如表 5.3 所示。

表 5.3　扩展考查人员列表

多多	贝贝	行行	小洁	姑姑	叔叔	毛毛	小岩

就是这样，先考查自己的好朋友，如果他们不是律师，就把他们的朋友加到考查列表的结尾，这意味着你将在自己的朋友、朋友的朋友中查找，使用这种算法将搜遍你的整个朋友圈，直到找到律师或者考查列表为空为止。

最终，小朗朗找到了好朋友贝贝的姑姑，她是一位大律师，并且解决了他的问题。

其实，存储所有考查对象的数组还有一个更专业的名字叫队列，队列是一种先入先出的数据结构。就像你去排队买咖啡，作为一个有素质的人，你会站到队尾。那么先来的人先买，后来的人后买。队列的工作原理与排队一样，先入队列的先出队列，后入队列的后出队列。

注意：队列只支持两种操作——从队尾入队和从队头出队。

刚刚使用的方法就是广度优先搜索算法。使用广度优先搜索算法可以解决是否有律师的问题。同时由于每次新增考查对象都是加入队列的结尾，最先找到的结果一定是关系更密切的朋友，这样一来，还能解决找到关系最近的律师的问题。也就是说，能判断律师是从一度朋友、二度朋友还是三度朋友中找到的。从这个例子中我们可以看出，广度优先搜索能够找出两个元素之间的最短距离。

5.2.2　问题求解

开始选课时，小朗朗只能选择没有先修课的科目，比如先选择数学，然后再选择计算机基础，这样一来，就可以学习 Java 了。为了找出没有先修课的科目，需要建立一个数组来记录每门课的先修课数量，将每门课的先修课数量初始化为 0，课程数量为 numCourses。

```
preListCount=[0]*numCourses
```

注意：Python 中的数组也叫列表，但为了方便理解，统一称为数组。

接下来，需要通过先修课的二维数组 preList 来计算每门课的先修课数量，代码如下：

```
01  for line in preList:                #取出二维数组中的每一行
02      for i in range(len(line)):      #针对每一行来为preListCount赋值
03          if line[i]==1:
04              preListCount[i]+=1
```

经过计算，可以得出每门课的先修课数量，如表 5.4 所示。

表 5.4　先修课数量

数学	计算机	Java	英语	算法
0	0	2	0	2

注意：同大部分语言一样，Python 的下标也是从 0 开始的，而不是从 1 开始的。

接下来，建立一个队列 canTake 存储目前可以选择的课程。将那些先修课数量为 0 的课程加入队列 canTake。

注意：队列是一种先入先出的数据结构，先加入队列的元素最先取出。

```
canTake=[]
for i in range(len(preListCount)):
    if preListCount[i] == 0:        #挑选先修课数量为 0 的科目
        canTake.append(i)
```

经过以上步骤，我们建立了存放每门课程先修课数量的数组 preListCount 和存放目前可以选修的课程的队列 canTake。接下来，小朗朗就可以使用广度优先遍历开始选课了。

广度优先遍历以队列为基础，首先加入直接可以选修的课程，依次选修队列里的每一门课程。学完这些课以后，再把那些新的可以选修的课程加入队列。直到队列为空，就可以把所有课程学完了。当然也有可能队列已经为空了，但是还有课没有学，这种情况是因为先修课图里面出现了环，使得这些课互为先修课。这种情况属于学校的课程设计有问题，这些课也就没办法学了。

首先，我们看一下广度优先遍历的代码模板是怎样的：

```
while len(canTake) !=0:
    thisClass=canTake[0]
    del canTake[0]                  #通过 del 删除队列的第 i 个元素
                                    #针对 thisClass 做处理
```

每次从队列中取出一门课，把它加入最终的选课列表。同时，如果哪门课程是以该课为先修课的，需要把它的先修课数量减 1。如果能够使某门课的先修课数量变为 0，就说明这门课也是可以选修的了，就把它也加入队列。有了以上的知识，就可以写出广度优先遍历的代码了：

```
classTake=[]
while len(canTake) !=0:
    thisClass=canTake[0]
```

```
    del canTake[0]                  #通过 del 删除队列的第 i 个元素
    classTake.append(thisClass)     #针对 thisClass 做处理
    for i in range(numCourses):
        if preList[thisClass][i] ==1: #preList 的第 thisClass 行为以 thisClass 为先修课的科目
            preListCount[i]-=1
            if preListCount[i] ==0: # 如果一门课的先修课为 0，就将其加入队列
                canTake.append(i)
```

到此为止，就完成了广度优先遍历解决选课问题的代码，小朗朗可以开心地上课了，来看一下代码 5.1 吧。

注意：通过以上方法得到的选课列表并不是唯一的。

5.2.3 最终代码

代码 5.1 利用广度优先遍历解决选课的烦恼

```
#广度优先遍历
def bfs(numCourses,preList):
    preListCount=[0]*numCourses
    for line in preList:                    #取出二维数组中的每一行
        for i in range(len(line)):          #针对每一行来为 preListCount 赋值
            if line[i]==1:
                preListCount[i]+=1
    canTake=[]
    for i in range(len(preListCount)):
        if preListCount[i] == 0:
            canTake.append(i)
    classTake=[]
    while len(canTake) !=0:
        thisClass=canTake[0]
        del canTake[0]                      #通过 del 删除队列的第 i 个元素
        classTake.append(thisClass)         #针对 thisClass 做处理
        for i in range(numCourses):
            if preList[thisClass][i] ==1:
                #preList 的第 thisClass 行为以 thisClass 为先修课的科目
                preListCount[i]-=1
                if preListCount[i] ==0:    # 如果一门课的先修课为 0，就将其加入队列
                    canTake.append(i)
    return classTake
```

其实，除了可以通过广度优先遍历来选课，还可以把这种办法应用到工程项目的规划，

甚至人生的规划中去。同时，这道题还是一个经典的拓扑排序问题，但这并不在本章的讨论范围内。

5.3　寻找制高点

又到了春暖花开的季节，学校组织同学们去爬香山。小朗朗和小伙伴们约定要走遍所有的制高点，他们找到了香山的地图开始分析要“占领”哪些山头。他们拿到的地图是一张海拔图，上面有每个地理位置的海拔高度数据。为了描述这张地图，我们同样使用一个 $M \times N$ 的二维数组来表示，每个点的海拔用数字来表示，如表 5.5 所示。

表 5.5　海拔图

1	3	2	3	5
3	4	5	6	3
2	7	4	3	3
5	2	2	3	1

那么，我们来看一下小朗朗想找的制高点是什么样的，制高点是可以一直向下到达东西南北四个方向的点。在向下走的过程中是不能爬坡的，也就是说，只能向比当前点低的点或者和当前点一样高的点的方向走。同时，走的过程中只允许向上下左右四个方向走，例如图中的 7 点、6 点和第 2 行第 3 列的 5 点就是制高点，它们可以通过图中带下画线的点走到四个边缘点。部分示例如表 5.6 和表 5.7 所示。

表 5.6　制高点图 1

1	3	2	3	5
3	4	5	6	3
2	7	4	3	3
5	2	2	3	1

表 5.7　制高点图 2

1	3	2	3	5
3	4	5	6	3
2	7	4	3	3
5	2	2	3	1

5.3.1　问题求解

这个问题其实是典型的搜索问题，拿到这个问题后，最开始想的肯定是从每个点出发来搜索是否能到达四个边缘。但是，不像迷宫问题，搜索的目标点不是一个单点，而是所有的边缘点，按照这种思路显然是效率很低的。

那么，如何来优化算法呢？这条思路不通，那么就换一个角度考虑问题，既然从每个点向边缘走效率会比较低，那么以边缘作为起点向内部开始遍历搜索，看下一个节点的高度是否高于或者等于自身的高度（就是可以从那点走到自身的点）。

然后标记那些能够到达的点为 True，继续搜索下去，直到不能走为止。按照同样的思路分别标记从四个边缘出发可以到达的点，那么最终四者均为 True 的点，就是我们寻找的制高点。这个问题是典型的逆向思维，有时候我们需要学会用逆向思维来思考问题。

为了标识地图上的每一个点，我们需要为这些点定义坐标。对于平面坐标来说，左上角的点为(0, 0)，右下角的点为(*m*-1, *n*-1)，如表 5.8 所示。

表 5.8　坐标系

（0，0）	（0，1）	（0，2）	（0，3）	（0，4）
（1，0）	（1，1）	（1，2）	（1，3）	（1，4）
（2，0）	（2，1）	（2，2）	（2，3）	（2，4）
（3，0）	（3，1）	（3，2）	（3，3）	（3，4）

有了上面的分析，就可以开始写代码了。首先，我们的已知数据就是这张地图，也就是一个二维数组。代码框架应该是这样的：

```
01   def solve(matrix):                #matrix 是存储地图的二维数组
02       if not matrix:
03         return matrix               #如果二维数组为空，那么就返回空
04       ......                        #书写相应逻辑代码
```

为了更好地表示可以移动的方向，同样使用一个二维数组来存储上下左右四个方向。这个技巧在以后的问题中会多次使用。

```
01 dir= [[0,1],[0,-1],[1,0],[-1,0]          #数组中四个元素代表四个方向
```

注意：[0,1]表示在二维坐标中，*x* 坐标+0，*y* 坐标+1，即向下移动。

为了方便理解，先定义两个变量来存储二维数组的大小：

```
m = len(matrix)              #表示二维数组有多少行
n = len(matrix[0])           #表示二维数组有多少列
```

接下来，以从山的北边上山为例来寻找制高点，有了第一个问题“选课的智慧”的基础后，我们知道队列是完成广度优先遍历必备的数据结构。所以，先把第一行的点全部加入队列，然后就可以开始进行广度优先遍历了：

```
topPoint = [(0, y) for y in range(n)]
```

由于从上下左右四个方向上山都是同一个子问题，因此最好把广度优先遍历写成一个函数，广度优先遍历的代码模板如下：

```
def bfs(set,m,n,matrix):          #set 为队列
  while len(set) >0:
      x,y=set.pop(0)              #取出队头的元素，由于是一个坐标，我们使用两个变量来接收
      #逻辑处理在这里
      if #可以到达:               #如果新的点符合要求，需要把它加入队列
          set.append(#新点)
```

广度优先遍历算法的步骤是从队头取出一个点，然后从该点出发，通过方向数组建立新的点，查询这些点是否可以到达。记录可以到达的点，同时把新点加入队列。为了保存那些可以到达的点，需要在广度优先遍历之前，建立一个队列的副本。在 Python 中，list()函数可以把将元组转换为列表，通过 list()函数建立 set 的一个副本，作为新的队列：

```
queue = list(set)
```

我们使用 queue 来做广度优先遍历，set 用来存储可以到达的点。广度优先遍历算法的代码如下：

```
def bfs(set,m,n,matrix):                                    #queue 为队列
  queue = list(set)          #list()函数创建 queue 为 set 的队列版本
  while len(queue) >0:
      x,y=queue.pop(0)       #取出队头元素，由于返回值是一个坐标，我们使用两个变量来接收
      for d in dir:
          nx=x+d[0]
          ny=y+d[1]
          if 0<=nx and nx <m and 0<=ny and ny<n:        #如果新的点在二维数组内
              if matrix[nx][ny] >= matrix[x][y]:        #如果新的点比原点高则可以走
                  if (nx, ny) not in set:               #如果新的点没有在原集合里
                      queue.append((nx, ny))            #在队列里添加新点，继续查找
                      set.add((nx, ny))                 #在集合里记录新点可以到达
```

经过上面的广度优先遍历，可以得到从某一个方向出发可以到达的点，那么我们执行四次，

就可以得到四个方向的结果了。为了后面的集合运算，我们使用 set()函数来把列表转换为集合：

```
topPoint = set([(0, y) for y in range(n)])          #上边的点的横坐标为 0，纵坐标为 0 到 n-1
leftPoint = set([(x, 0) for x in range(m)])         #左边的点的横坐标为 0 到 m-1，纵坐标为 0
bottomPoint = set([(m-1, y) for y in range(n)])     #下边的点的横坐标为m-1,纵坐标为0到n-1
rightPoint = set([(x, n-1) for x in range(m)])      #右边的点的横坐标为0到m-1,纵坐标为n-1
bfs(topPoint, m, n, matrix)                         #依次调用 4 次 bfs
bfs(leftPoint, m, n, matrix)
bfs(bottomPoint, m, n, matrix)
bfs(rightPoint, m, n, matrix)
```

注意：set()函数接收一个列表，输出一个集合。

从四个方向上山，可以得到四个集合，分别为：

```
top {(0, 1), (1, 2), (0, 0), (1, 3), (1, 0), (2, 1), (1, 1), (0, 4), (0, 3), (0, 2)}
left {(0, 1), (1, 2), (0, 0), (1, 3), (3, 0), (2, 1), (2, 0), (1, 0), (1, 1)}
bottom {(1, 2), (3, 2), (1, 3), (3, 3), (3, 0), (2, 2), (3, 1), (1, 4), (2, 1), (2,
   3), (0, 4), (3, 4), (2, 4)}
right {(1, 2), (1, 3), (3, 3), (2, 2), (1, 4), (2, 1), (2, 3), (0, 4), (3, 4), (2, 4)}
```

集合和数组很像，它和数组的区别是集合内部没有重复的数据。根据上面的分析，最终的制高点应该为四个集合的交集，也就是说，从哪个方向都可以一直向上的上山的点。

```
result=topPoint & leftPoint & bottomPoint & rightPoint
```

再经过交集的计算就可以到最终的制高点了，一共有三个制高点。

```
{(1, 2), (1, 3), (2, 1)}
```

5.3.2　集合

上面程序中&的作用就是求集合的交集。使用交集运算，省去了循环遍历四个数组来寻找相同的点的过程。只需要一个语句，就可以得到结果，非常神奇！它是怎么得到结果的呢？假如你早饭吃了包子、粥和鸡蛋，午饭吃了包子、烤鸭、臭豆腐，那么早饭和午饭这两个集合的交集就是包子，如图 5.7 所示。

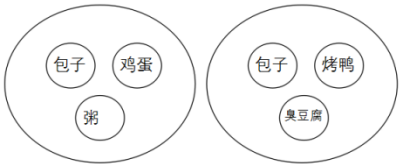


图5.7　集合图

5.3.3 最终代码

综合上面的代码，可以形成最终的程序，如代码 5.2 所示。

代码 5.2 利用广度优先遍历解决制高点问题

```
01 bfs(set,m,n,matrix):          # queue 为队列
02     dir = [[0, 1], [0, -1], [1, 0], [-1, 0]]  # 数组中四个元素代表四个方向
03     queue = list(set)         # list()函数创建 queue 为 set 的队列版本
04     while len(queue) > 0:
05         x, y = queue.pop()    # 取出队头的元素，由于是一个坐标，我们使用两个变量来接收
06         for d in dir:         # 循环遍历 4 个方向
07             nx = x + d[0]
08             ny = y + d[1]
09             if 0 <= nx and nx < m and 0 <= ny and ny < n: # 如果新的点在二维数组内
10                 if matrix[nx][ny] >= matrix[x][y]:       # 如果新的点比原点高则可以走
11                     if (nx, ny) not in set:
12                         queue.append((nx, ny))
13                         set.add((nx, ny))
14
15 def solve(matrix):             # matrix 为存储地图的二维数组
16     if not matrix:
17         return matrix         # 如果二维数组为空，那么就返回空
18     m = len(matrix)           # 表示二维数组有多少行
19     n = len(matrix[0])        # 表示二维数组有多少列
20     topPoint = set([(0, y) for y in range(n)])     # 上边的点的横坐标为 0，纵坐标为 0 到 n-1
21     leftPoint = set([(x, 0) for x in range(m)])    # 左边的点的横坐标为 0 到 m-1，纵坐标为 0
22     bottomPoint = set([(m - 1, y) for y in range(n)]) # 下边的点的横坐标为 m-1，纵坐标为 0 到 n-1
23     rightPoint = set([(x, n - 1) for x in range(m)]) # 右边的点的横坐标为 0 到 m-1，纵坐标为 n-1
24     bfs(topPoint, m, n, matrix)                    # 依次调用 4 次 bfs
25     bfs(leftPoint, m, n, matrix)
26     bfs(bottomPoint, m, n, matrix)
27     bfs(rightPoint, m, n, matrix)
28     result = topPoint & leftPoint & bottomPoint & rightPoint   #求集合的交集
29     return result
30 matrix=[[1, 3, 2, 3, 5],                          #测试程序
31        [3, 4, 5, 6, 3],
32        [2, 7, 4, 3, 3],
33        [5, 2, 2, 3, 1]]
34 s=solve(matrix)
35 print(s)
```

广度优先遍历本质上是查找算法。为了解决查找问题，之前我们介绍了很多查找算法，例如顺序查找、二分查找、哈希查找。广度优先遍历更多地针对图的搜索，除此之外，广度优先

搜索可以解决单一起点的最短路径问题，图和最短路径问题在最后一章会有详细讲解。

5.4　合法的括号

小朗朗学校的计算机课正在学习 racket 编程语言，这门语言是函数式编程语言。作为教学语言，它能够让我们学习函数式编程。所谓函数式编程，就是几乎所有的语句都是通过函数来完成的，例如在 racket 里没有循环，循环是通过递归调用函数来实现的，也就是说，代码里面充满了括号。老师每次在判作业的时候，都发现同学对于括号的运用没有掌握好，总是会有左右括号不匹配的问题。

注意：函数是一个功能代码的集合，方便人们反复调用。

下面的代码就是一小段 racket 编程语言的代码，我们不用理解它的含义。

```
01 (define (rainbow p)
02 (map (lambda (color)
03  (colorize p color))
04  (list "red" "orange" "yellow" "green" "blue" "purple")))
05 (rainbow (square 5))
```

5.4.1　问题求解

为了帮助老师更快地修改学生的括号匹配问题，小朗朗决定开发一个小程序，来寻找正确的小括号组合。我们对包含括号的合法的字符串并不陌生，字符串中的左右括号数应该相同，而且每个右括号左边一定有其对应的左括号。给定一个包含括号的字符串，那么问题就变成了在一个给定的输入字符串中，移除掉最少量的错误括号，从而使得这个字符串变为有效的字符串，并且返回所有有效的字符串。

注意：字符串中除了（和）之外，还有字母和数字。

对于括号匹配的问题，最先想到的应该是用栈来解决。常规的处理操作是，在遇到左括号的时候入栈，遇见右括号的时候出栈。但是，我们知道对于最短路径，广度优先遍历算法不仅可以找到合法字符串，还可以找到经过最少的变化就能得到的合法字符串，所以广度优先遍历算法更适合解决这个问题。

举一个例子，有一个学生写了这样一个字符串：(a)(b))()，我们的程序应该能够识别最长的合法字符串有两个：(a)(b)()和(a(b))()。分别是由去掉图 5.8 所示的下画线标注的括号得到的。

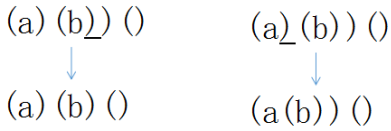


图5.8　合法字符串示例1

对于一个字符串，在任何时候如果右括号的个数大于左括号的个数，则说明从开始到现在的位置肯定可以删除若干个右括号。如果这个子字符串包含多个右括号，该删除哪一个呢？当然是删除任何一个都可以。对于上面这个例子来说，从开头到第三个右括号的位置多了一个右括号，因此我们需要删掉一个，分别删除这三个右括号，但是只会产生两个结果，因为删除第二个右括号和第三个右括号会得到同样的结果。

在删除括号的时候，为保证不会产生重复值，需要记录之前的结果，避免出现重复。其实还有更好的办法，这里先留一个悬念，之后我们再讲解如何防止重复。这样，就可以使当前这个子字符串不再包含多余的右括号了。对于(ab)()，虽然也是合法字符串，但是它是通过删除 3 个左括号或右括号得到的，并不是最长的合法字符串，所以不是我们想要的结果。

还有一种情况是包含了多余的左括号，可以使用同样的方法去除多余的左括号。例如，输入是(a)(b))(，那么程序应该能够识别最长的合法字符串为(a)(b)，如图 5.9 所示。

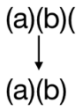


图5.9　合法字符串示例2

首先，解决什么样的字符串是合法的字符串的问题。为了判断一个字符串是否合法，就像上面提到的，可以使用一个栈，这也是栈这种数据结构的一个典型应用。

为了简化操作，还可以使用一个变量来模拟栈，从头到尾地扫描字符串，只要遇到左括号就让变量加 1，遇到右括号就让变量减 1，一旦发现在这个过程中，某一步使得该变量的值小于 0，就说明该字符串是一个非法字符串，如果全部都通过的话，就说明它是合法字符串。这种只用一个计数变量的方法不仅容易理解，同时相比于传统的栈的方法，能够节省很多内存开销。

```
01 def isvalid(str):
02   count= 0                          #创建一个变量记录括号的数量
03   for c in str:
04       if c == '(':                  #遇到左括号就让变量加 1
05             count += 1
06       elif c == ')':
```

```
07            count -= 1                #遇到右括号就让变量减 1
08            if count < 0:             #减 1 以后要注意会不会出现变量变为负数的情况
09             return False
10    return count == 0
```

注意：错误情况包括左右括号数量不相等，左右括号顺序不对。

我们从最开始的字符串出发分析这个问题。使用广度优先遍历算法，一定少不了队列，二话不说，先把初始字符串加入队列再说。

我们每次从队列中取出一个字符串，查看它是不是合法的，如果它是合法的，则把它加入结果集，返回结果就好了。如果不合法，我们该怎么办呢？

其实很简单，如果这个字符串不合法，遍历这个字符串，只要遇到左右括号字符的时候，我们就去掉该括号字符生成一个新的字符串，把它也加入到队列中进行分析。对于括号数量为 N 的字符串，理论上，这一步要产生 $N-1$ 个子字符串，当然还要去除重复的数据，可能会小于 $N-1$ 个子字符串，如图 5.10 所示。

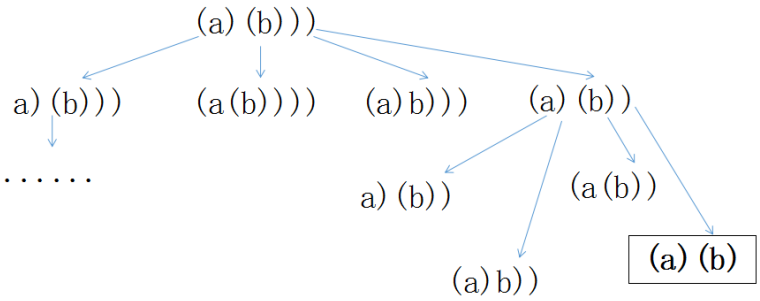


图5.10　字符串的层级关系

为了减少重复的计算，可以把字符串放到一个哈希集合里，在上一个问题中，我们知道集合是不存储相同的值的，在这里，我们同样利用这个性质来去重。

注意：每个生成的子节点就是对每个位置移除左括号或者右括号剩下的子字符串。

这个问题要求去除最少的括号来得到给定字符串的合法的含有括号的字符串，我们需要找出所有合法的结果，还要保证去除最少的括号，要同时满足这两个条件。

找出所有合法的结果需要我们一直找到队列为空为止。而保证去除最少的括号，需要当发现了合法字符串以后，就不要再分析比它更短的字符串了，这就需要一层一层地分析。如果发现了合法的字符串，就不需要继续搜索下去了，直接跳出循环，这里我们使用的技术叫作剪枝。

注意：在搜索过程中，所谓剪枝是通过某种判断，避免一些不必要的遍历过程，就是剪去搜索树中的某些枝条。

对队列中的每个字符串都进行相同的操作，还有一种情况是直到队列为空还没找到合法的字符串，那就返回空集。

我们使用集合来去除重复的临时数据，最终的结果也需要去除重复的数据，可以使用上一个问题中的 set()函数来帮忙，代码如下：

```
01 queue= [str]              #先把初始字符串加入队列
02 queue=list(set(set))      #去除重复数据的方法
```

注意：set()函数和 list()函数是两个非常重要的函数，分别用于创建集合和列表以及进行相应的转换。

你对广度优先遍历的算法模板一定很熟悉了，来看下面的代码：

```
01 while len(queue)>0:          #队列不为空的时候就开始广度优先遍历
02   s=queue.pop()              #队列头元素出队列
03   #业务逻辑写在这里
```

在每一层的分析中，我们都需要把队列中合法的字符串加到最终结果的集合中，为此，还需要创建一个数组存储最终的结果。

```
01 res=[]                    #res 存放最终结果
```

接下来，就可以使用广度优先遍历算法对队列进行分析了。需要对队列中的每个元素进行判断，考查它是否为合法字符串。和以往不一样的地方在于，分析完一个数据后不是立即把它删除，而是留下来做之后的数据处理，这样设计的原因在于算法是一层一层地进行分析的。这样一来就需要在发现合法字符串以后就结束循环，而在本层分析中没有发现合法字符串时才生成下一层字符串。代码如下：

```
01 def bfs(str):
02     res=[]                              #res 存放最终结果
03     queue= [str]                        #先把初始字符串加入队列
04     while len(queue)>0:                 #队列不为空的时候就开始进行广度优先遍历
05         for i in range(len(queue)):     #依次从队列中取出字符串开始分析
06             if isvalid(queue[i]):
07                 res.append(queue[i])    #发现合法字符串就放到结果集里
08         if len(res)>0:
09             return list(set(res))       #如果发现合法字符串就结束循环返回结果，并用集合去重
10     #生成下一层的队列数据
11     return list(set(res))               #先把数组转换为集合，再转换回数组来去重
```

由于采用了层级分析，所以每一轮 for 循环处理的字符串都是等长的字符串，如果没有发现合法的字符串才会继续分析下一轮长度减 1 的字符串。为了得到下一层的字符串，可以从队列中拿出每个字符串，按照之前的分析，只要遇到左右括号字符，就去掉该括号字符生成一个新的字符串，把它也加入到队列之中。

```
01 temp=[]                          #建立临时结果集
02 for s in queue:                  #取出队列中的每一个字符串
03    for i in range(len(s)):       #对于每个字符串，分别查看每个字符
04       if s[i]=='('or s[i]==')': #如果为左右括号则生成新的字符串，并将其加入临时结果集
05          temp.append(s[:i] + s[i + 1:])
06 queue=list(set(temp))            #把新生成的字符串赋值给队列，并用 set()函数去重
```

5.4.2　最终代码

整合上面的代码，我们来看完整的代码，如代码 5.3 所示。

代码 5.3　利用广度优先遍历解决括号合法性问题

```
01 def isvalid(str):
02  count= 0                        #创建一个变量记录括号的数量
03  for c in str:
04      if c == '(':                #遇到左括号就让变量加 1
05          count += 1
06     elif c == ')':
07         count -= 1               #遇到右括号就让变量减 1
08         if count < 0:            #减 1 以后要注意会不会出现变量变为负数的情况
09          return False
10 return count == 0
11 def bfs(str):
12    res=[]                        #res 存放最终结果
13    queue= [str]                  #先把初始字符串加入队列
14    while len(queue)>0:           #队列不为空的时候就开始进行广度优先遍历
15       for i in range(len(queue)):        #依次从队列中取出字符串开始分析
16          if isvalid(queue[i]):
17             res.append(queue[i])         #发现合法字符串就放到结果集里
18       if len(res)>0:
19          return list(set(res)) #如果发现合法字符串就结束循环返回结果，并用集合去重
20       temp=[]                   #建立临时结果集
21       for s in queue:           #取出队列中的每一个字符串
22          for i in range(len(s)):         #对于每个字符串，分别查看每个字符
23              if s[i]=='('or s[i]==')': #如果为左右括号则生成新的字符串，并将其加入临时结果集
24              temp.append(s[:i] + s[i + 1:])
```

```
25          queue=list(set(temp))                #把新生成的字符串赋值给队列，并用 set()函数去重
26      return list(set(res))                    #先把数组转换为集合，再转换回数组来去重
```

这个问题是使用广度优先搜索算法来解决的，但是又和之前的问题不太一样。在之前的问题中，每处理完一个数据就把它从队列中去除，同时，按照某种特定的规则来寻找新的数据再加入到队列之中。而在本题中，为了能够得到所有的最长的合法字符串，我们采用一层一层分析的方式，为此需要保留本层数据以备生成下一层数据。同时，在本例中，我们再次使用了集合来解决重复问题。广度优先搜索算法的变化很多，核心就是由近到远地寻找目标元素。

5.5 树的右侧

到了秋天，采摘园里的柠檬都熟了，看着这硕果累累的景象，小朗朗和小伙伴贝贝相约去采摘园摘柠檬。为了提高采摘速度，小朗朗和贝贝约定每人采摘每棵树一半的柠檬，同时为了可持续发展，只采摘柠檬树最外层的柠檬。

假如柠檬树的每个树干只有两个树枝、一个树枝、零个树枝三种可能，那么一棵树可以被看作一棵二叉树，小朗朗的任务是摘树的右侧的柠檬，让我们来帮小朗朗数一下，他可以摘到哪些柠檬吧。

5.5.1 问题求解

这个问题可以用二叉树来表示，如图 5.11 和图 5.12 所示。

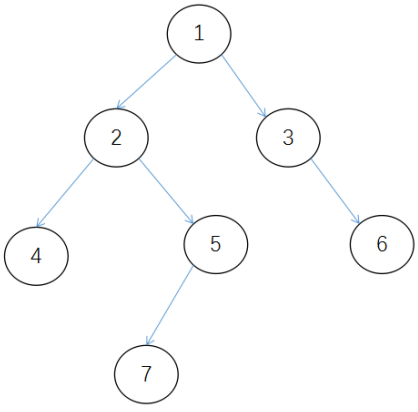


图5.11　柠檬树1

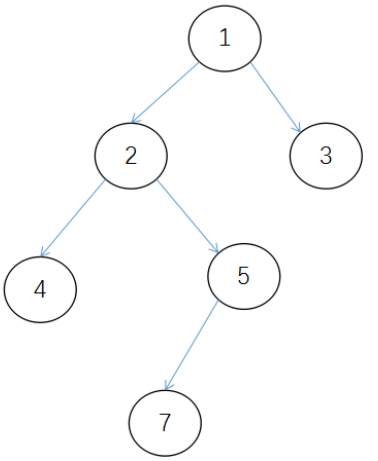


图5.12　柠檬树2

对于上面的柠檬树 1 来说，小朗朗可以摘到的柠檬为 1、3、6、7 这 4 个柠檬。对于柠檬树 2 来说，小朗朗可以摘到的柠檬为 1、3、5、7，也就是说要摘下站在柠檬树右边能看到的柠檬。

注意：二叉树是每个节点最多有两个子树的树结构。

那么，此时，本题的问题变成了给定一棵二叉树，想象你自己站在该二叉树的右侧，按照从根节点到底端叶子节点的顺序，找出可以获取的节点的值的问题。

为了表示这样的柠檬树，需要定义一个二叉树的结构，代码如下：

```
01 class TreeNode(object):      #定义二叉树的根节点为 TreeNode
02   def __init__(self, x):
03       self.val = x           #定义二叉树的每个元素的值，这里表示柠檬的编号
04       self.left = None       #定义二叉树的每个元素的左子树
05       self.right = None      #定义二叉树的每个元素的右子树
```

根据前面的分析，为了找出二叉树右侧的节点的值，我们需要一层一层地取出每一层的数据，然后把每层的数据放入最终结果中。我们为每个节点设定一个层级，比如根节点的层级是第 1 层，根节点的子节点的层级是第 2 层，再往下的节点的层级依次递增，如图 5.13 所示。

对于树的遍历问题，我们通常会使用广度优先遍历算法或者深度优先遍历算法，对于这个问题，该选择哪一种算法呢？由于最终的结果是每一层节点的最后一个节点，那么使用广度优先遍历会更好一些。

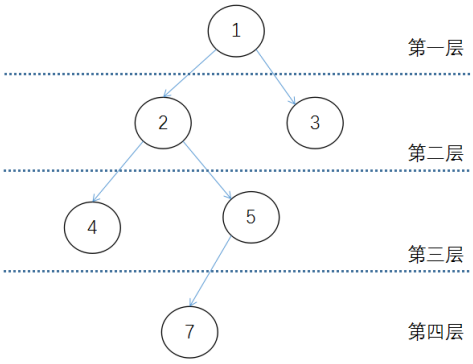


图5.13　二叉树的层次结构

这个题目同样是遍历问题，或者说是求二叉树广度优先遍历的一种变形，我们只需要保存每一层最右边的数字即可。既然是广度优先遍历算法，那么一定要用到队列，使用队列来保存待处理的节点。首先处理根节点，存储根节点的值，同时建立存放结果的列表，代码的框架如下：

```
01 def rightSide(root):
02   result = []            #存放结果数据
03   queue = [root]         #队列存放待处理数据
04   while len(queue)>0:
05        #队列不为空时，依次取出数据做处理
06   return result
```

使用广度优先遍历算法的时候，需要先取出每层的节点，然后处理该节点，之后把下一层的左右子节点都存入队列，依此类推，到队列为空，处理就完毕了。

那么本题的核心问题是如何找到每层的最后一个节点，为了解决这个问题，可以在处理每层数据的时候记录一下当前层的节点数 *N*，那么本层最后一个节点就是当前队列里的第 *N* 个数据。在寻找每层最后一个元素的同时，把每个节点的左右节点加入队列，一直到这一层的所有元素都分析完为止。对于每层节点的分析，使用一个 for 循环来处理。整个程序在队列为空时结束：

```
01 while  len(queue)>0:              #队列不为空时，依次取出数据做处理
02    length = len(queue)            #先保存本层的数量
03    for i in range(length):        #依次处理每一行
04       node=queue.pop()            #取出队列头的点
05       if i==length-1:             #当该点是最后一个点的时候把它加入结果数组里
06          result.append(node.val)
07       if node.left !=None:
08          queue.append(node.left)          #把左节点加入队列
```

```
09        if node.right !=None:
10            queue.append(node.right)          #把右节点加入队列
```

注意：数组下标从 0 开始，所以代码中每层最后一个元素是 N-1。

在上面的例子中，通过广度优先遍历就可以自顶向下找到这些具有下画线的节点，每条线上的数字代表访问它们的顺序。如图 5.14 所示。

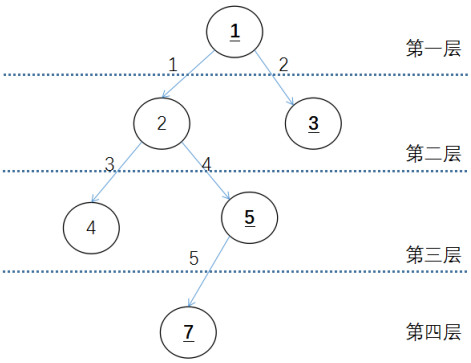


图5.14　二叉树的广度优先遍历

5.5.2　最终代码

整合上面的代码，我们来看完整的代码，如代码 5.4 所示。

代码 5.4　利用广度优先遍历解决树的右侧问题

```
01 class TreeNode(object):              #定义二叉树的根节点为 TreeNode
02    def   init   (self, x):
03        self.val = x                  #定义二叉树的每个元素的值，这里表示柠檬的编号
04        self.left = None              #定义二叉树的每个元素的左子树
05        self.right = None             #定义二叉树的每个元素的右子树
06
07 def rightSide(root):
08    result = []                       #存放结果数据
09    queue = [root]                    #队列存放待处理数据
10    while len(queue)>0:               #队列长度不为空则继续循环
11        length = len(queue)           #获取队列长度
12        for i in range(length):       #依次处理每一行
13            node=queue.pop()          #取出队列头的点
14            if i==length-1:           #当该点是最后一个点的时候把它加入结果数组里
15                result.append(node.val)
16            if node.left !=None:
```

```
17                queue.append(node.left)      #把左节点加入队列
18            if node.right !=None:
19                queue.append(node.right)     #把右节点加入队列
20     return result
21
22 node1=TreeNode(1)                           #以下为测试代码
23 node2=TreeNode(2)
24 node3=TreeNode(3)
25 node1.right=node2
26 node2.right=node3
27
28 print(rightSide(node1))
```

当然，本题还有其他解法。比如，可以改变遍历的顺序，即按先访问当前节点，再访问右节点，最后访问左节点的顺序。每次都将节点放进队列，但是在取节点的时候，只需记录第一个节点即可，其他的都可以释放掉，这个方法留给读者自己去实现。

第 6 章

回溯算法

回溯算法可以看成走迷宫。我们不知道出口在那里，所以只能不断地深入，尝试不同的路线。但一旦找到了出口便可以回溯到起点，辨清路线。本章主要涉及的问题有：

- 遍历所有排序方式
- 经典问题的组合
- 查找单词问题
- 八皇后问题
- 教你解数独

6.1 什么是回溯

简单来讲，回溯采用试错的方法解决问题。一旦发现当前步骤失败，回溯方法就返回上一个步骤，选择另一种方案继续试错。

如图 6.1 所示，小朗朗在没有地图的情况下需要从迷宫入口出发到达出口。小朗朗走迷宫的方案和回溯的想法一致。他想，不碰墙不回头，碰了墙就退一步，换另一个方向接着走，这样最后肯定会到达终点。

小朗朗从入口出发向前走，每遇到岔路口都向左转。在第二个岔路口的左分支小朗朗碰到了墙。这时，他退到第二个岔路口，继续向前走到第三个岔路口。在第三个岔路口，小朗朗选

择左分支后碰墙，退回，选择右分支，顺利到达出口。

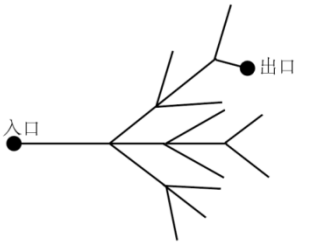


图6.1　迷宫

小朗朗在没有尝试所有路线的情况下就找到了正确路线，可见回溯算法的一个优点是搜索速度快。当然，如果出口恰好在最后一个分支的底端，那么小朗朗的方案就没有特别的优势了。

回溯算法又称为试探法，它的主要思想我们已经通过走迷宫的例子讲解了。回溯算法针对的大多数问题有以下特点：问题的答案有多个元素（可想象成走迷宫时做的多个决定）、答案需要满足一些约束（比如数独）、寻找答案的方式在每一个步骤相同（用同样的方案对待每一个岔路口）。回溯算法逐步构建答案，并在确定候选元素不满足约束后立刻放弃候选元素（一旦碰墙就返回），直到找到答案的所有元素。

接下来，我们来看几个回溯算法的问题。

6.2　遍历所有排序方式

小朗朗最近有 4 本想读的书:《红色的巴黎》《黄色的梧桐树》《蓝色的夏天》《绿色的天空》。如果小朗朗一次只能从图书馆借一本书，他一共有多少种借书的顺序呢？如果学过统计学的话，你会知道 4 本书一共有 4！=24 种不同的排序方式。但是，我们不只想知道这个总数，还想一一输出所有的排序方式。

图6.2　小朗朗想读的4本书

6.2.1　问题求解

我们来模拟一下一般是怎么排序的，如图 6.3 所示。我们把 4 本书放入 4 个空位中。首先，

第一个空位有 4 个选择：红、黄、蓝、绿。

假设第一本书是《红色的巴黎》，如图 6.4 所示，那么第二本书剩下 3 个选择：黄，蓝，绿。

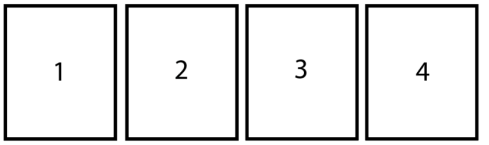


图6.3　排位顺序

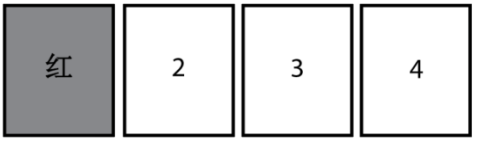


图6.4　假设第一本书

假设第二本书是《黄色的梧桐树》，如图 6.5 所示，那么第三本书剩下两个选择：蓝，绿。

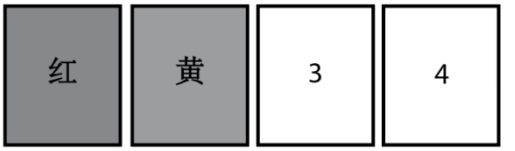


图6.5　假设第二本书

最后，假设第三本书是《蓝色的夏天》，如图 6.6 所示。那么第四本书只能是《绿色的天空》，如图 6.7 所示。

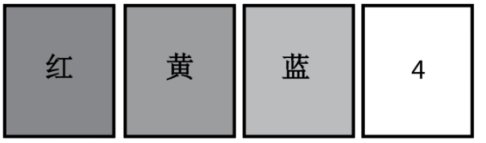


图6.6　假设第三本书

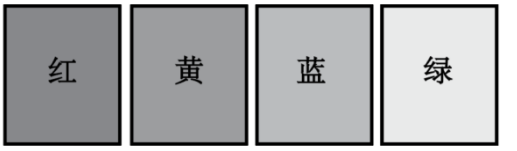


图6.7　假设第四本书

这是 24 种排序方式中的一种。我们也可以使第一本书是《黄色的梧桐树》，或者第二本书是《蓝色的夏天》，从而得出不同的排序方式。注意，第一本书有 4 种选择，第二本书有 3 种选择，第三本书有 2 种选择，第四本书只有 1 种选择，因为每一次都要把上一本书从我们的选择中剔除。

按照上面的解题思路，如果不用回溯可以这样写代码：

```
01 def solvePermutationBadWay(self,array):       #array 是一个列表,这里是[红，黄，蓝，绿]
02   solution=[]                                      #创建一个空的 list
03   for i in range(len(array)):                      #第一本书有 4 种选择
04      newSolution1 = solution+[array[i]]            #创建新的 list，把第一本书加进去
05      newArray1 = array[:i] + array[i + 1:]         #newArray1 把第一本书剔除
06      for i in range(len(newArray1)):               #第二本书剩下 3 种选择
07         newSolution2 = newSolution1 + [newArray1[i]] #创建新的 list，把第二本书加进去
08         newArray2 = newArray1[:i] + newArray1[i + 1:]  #newArray2 把第一本和第二本书剔除
```

```
09          for i in range(len(newArray2)):          #第三本书剩下 2 种选择
10              newSolution3 = newSolution2 + [newArray2[i]]  #创建新的 list，加入第三本书
11              newArray3 = newArray2[:i] + newArray2[i + 1:]
                                                     #newArray2 把第一、二、三本书剔除
12              for i in range(len(newArray3)):      #第四本书剩下 1 种选择
13                  newSolution4 = newSolution3 + [newArray3[i]]        #把第四本书放进去
14                  print (newSolution4)             #输出排序方式
```

如果这样写代码你会发现有一个问题，那就是你需要提前知道数组的长度。另外一个问题就是代码十分累赘。

```
01 def solvePermutationBadWay(self,array):          #array 是一个列表,这里是[红，黄，蓝，绿]
02   solution=[]                                                #创建一个空的 list
03   for i in range(len(array)):                          #第一本书有 4 种选择
04     newSolution1 = solution+[array[i]]                 #创建新的 list，把第一本书加进去
05     newArray1 = array[:i] + array[i + 1:]              #newArray1 把第一本书剔除
06     for i in range(len(newArray1)):                    #第二本书剩下 3 种选择
07       newSolution2 = newSolution1 + [newArray1[i]]     #创建新的 list，把第二本书加进去
08       newArray2 = newArray1[:i] + newArray1[i + 1:]    #newArray2 把第一本和第二本书剔除
09       for i in range(len(newArray2)):                          #第三本书剩下 2 种选择
10         newSolution3 = newSolution2 + [newArray2[i]]           #创建新的 list，加入第三本书
11         newArray3 = newArray2[:i] + newArray2[i + 1:]          #newArray2 把第一、二、三本书剔除
12         for i in range(len(newArray3)):                                  #第四本书剩下 1 种选择
13           newSolution4 = newSolution3 + [newArray3[i]]                   #把第四本书放进去
14           print (newSolution4)                                           #输出排序方式
```

图6.8　重复的代码

观察一下图 6.8，代码是一环套一环的，每一环都是重复的。排第一本书的时候，有 4 种选择，答案集合就是这 4 种选择在各结尾增加剩下的三本书的排序集合。那么，只需要得出剩余的三本书的排序集合就可以了。

同样的，排这三本书时第一本书有三种选择，所以只需要得出剩余的两本书的排序集合就可以了。那么，推到最后，我们排两本书时，只需要知道最后一本是什么就可以了。

其实就是一个大的问题套着中的问题再套着小的问题，问题都是一样的：输出排序集合。唯一不同的是书本和书本的数量。

6.2.2　最终代码

我们定义一个 helper 方法来帮助概括这些重复的代码。代码 6.1 所示的是 helper 方法的完整代码，下面我们会详细进行介绍。

代码 6.1　helper 方法

```
01    def helper(self,array,solution):
```

```
02        if len(array)==0:                    #如果没有剩余书籍
03            print(solution)                  #输出结果
04            return                           #返回上一级
05        for i in range(len(array)):          #遍历书籍
06            newSolution = solution+[array[i]]    #新建排序列表
07            newArray = array[:i]+array[i+1:]     #新建书籍列表
08            self.helper(newArray,newSolution)    #回溯
```

我们先来看一下 helper 方法的参数：

- array，存储当前排序对象的列表，比如[红，黄，蓝，绿]，[黄，蓝，绿]，[蓝，绿]，[绿]。
- solution，存储当前排序组合的列表，比如[红], [红，黄], [红，黄，蓝], [红，黄，蓝，绿]。

接下来我们看一下 helper 方法履行的任务。

1. 把第一本书的选择加入排序列表。

```
def helper(self,array,solution):
        for i in range(len(array)):                  #第一本书的选择
            newSolution = solution+[array[i]]        #加入书本
```

注意：我们这里的“第一本书”不一定指 4 本书中的第一本，而是代指剩余书籍中的“第一本书”。

第一次调用 helper 方法时，排第一书有 4 种选择，那么依次把这 4 个选择加入排序列表。所以，新的排序列表是[红]，[黄]，[蓝]，[绿]。
第二次调用 helper 方法时，排第一书我们有 3 种选择，仍然依次把这三本书加入列表。所以，新的排序列表是[红，黄]，[红，蓝]，[红，绿]，[黄，红]，[黄，蓝]，[黄，绿]，[蓝，红]，[蓝，黄]，[蓝，绿]，[绿，红]，[绿，黄]，[绿，蓝]。

注意：每加一本书就需创建新的排序列表，因为不同的选择不能同时存在，任一本书当第一本书都是一种新的排序方式。

2. 把加入的书本从 array 中删除。
书已经被排过就不再是一个选择了。我们借的第一本书不再是第二本书的选择之一，所以需要把第一本书从排序对象列表中删除。

```
04            newArray = array[:i]+array[i+1:]           #删除书本
```

需要创建一个新的对象列表，因为每一次我们删除的书籍都是不一样的。

3. 寻找剩余书本的排序组合。

```
05          self.helper(newArray,newSolution)      #寻找剩余对象的排序集合
```

找出第一本书的选择后，只要找到剩余书本的排序集合，把它们加到第一本书后面就可以了。也就是说，需要再次调用 helper 方法，传入新的排序集合和新的对象集合。这样，下一次的“第一本书”其实就是这次的“第二本书”。

4. 检查是否得出排序方式。
如果不加上一个终止条件的话，helper 方法不会知道什么时候输出结果。所以需要在一开始就加上 if 判断条件，如果没有排序对象了就输出结果，并返回上一次调用它的地方。

```
01 def helper(self,array,solution):
02       if len(array)==0:                  #如果没有剩余排序对象
03          print(solution)                 #输出结果
04          return                          #返回上一次被调用的地方
```

返回上一级后，因为最后一本书只有一种选择，所以 helper 方法会再次返回上一次调用它的地方，也就是排倒数第二本书的地方。走完第二本书的 for 循环后，会返回到倒数第三本书的地方，依次类推，一直到第一本书。

有了 helper 方法，只需要调用它就可以解决排序问题了。

```
01 def solvePermutation(self,array):
02   self.helper(array,[])              #调用 helper 方法
```

注意：因为我们还没有开始排序，所以第一次调用 helper 方法时传入的排序列表为空。

下面来看一下完整的代码，如代码 6.2 所示。

代码 6.2　用回溯算法解决排序问题

```
01 class Solution():
02   def solvePermutation(self,array):
03      self.helper(array,[])                    #调用 helper 方法
04   def helper(self,array,solution):
05      if len(array)==0:                        #如果没有剩余排序对象
06         print(solution)                       #输出结果
07         return                                #返回上一次被调用的地方
08      for i in range(len(array)):              #第一本书的选择
09         newSolution = solution+[array[i]]     #加入书本
10         newArray = array[:i]+array[i+1:]      #删除书本
11         self.helper(newArray,newSolution)     #寻找剩余对象的排序集合
```

来试一下排序小朗朗的 4 本书：

```
Solution().solvePermutation(["红","黄","蓝","绿"])
```

以下是输出结果：

```
['红', '黄', '蓝', '绿']
['红', '黄', '绿', '蓝']
['红', '蓝', '黄', '绿']
['红', '蓝', '绿', '黄']
['红', '绿', '黄', '蓝']
['红', '绿', '蓝', '黄']
['黄', '红', '蓝', '绿']
['黄', '红', '绿', '蓝']
['黄', '蓝', '红', '绿']
['黄', '蓝', '绿', '红']
['黄', '绿', '红', '蓝']
['黄', '绿', '蓝', '红']
['蓝', '红', '黄', '绿']
['蓝', '红', '绿', '黄']
['蓝', '黄', '红', '绿']
['蓝', '黄', '绿', '红']
['蓝', '绿', '红', '黄']
['蓝', '绿', '黄', '红']
['绿', '红', '黄', '蓝']
['绿', '红', '蓝', '黄']
['绿', '黄', '红', '蓝']
['绿', '黄', '蓝', '红']
['绿', '蓝', '红', '黄']
['绿', '蓝', '黄', '红’]
```

6.3　经典问题的组合

小朗朗这学期想上两门选修课，他有 4 种选择：A.微积分，B.音乐，C.烹饪，D.设计。小朗朗一共有多少种不同的选课组合呢？用数学方法很容易得出一共有 $C_4^2 = 6$ 种不同的选课组合，不过，我们想用计算机把这些组合一一输出。

6.3.1　问题求解

如图 6.9 所示，有两个空位，4 个选项。下面模拟一下选课，以便找到解题思路。

首先，第一次选课的时候有 4 种选择，A、B、C、D。假设选择 A，那么，第二次选课时

剩余三个选择：B、C、D，如图 6.10 所示。

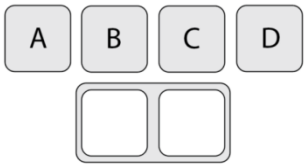


图6.9　选课

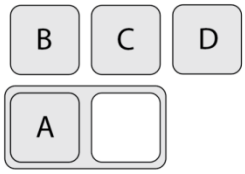


图6.10　第一次选课

假设选择了 B，得出的是 AB 这个选课组合，如图 6.11 所示。

再来看看其他的组合。这次我们首先选择 B。在第二次选课的时候，我们的选项中没有 A，因为从 A 开始的时候我们已经得出 AB 这个选课组合了。所以，第二次选课我们的选项是：C、D，如图 6.12 所示。

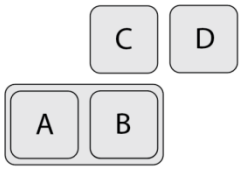


图6.11　第二门课选择B

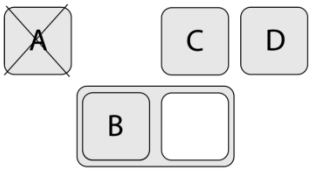


图6.12　第一次选择B

假设第二门课选择了 C，得到 BC 这个选课组合，如图 6.13 所示。

同样的，我们也可以从 C 和 D 开始。从 C 开始的话，第二门课只有 D 一个选项。从 D 开始的话，A、B、C 都不再是选项，如图 6.14 所示。

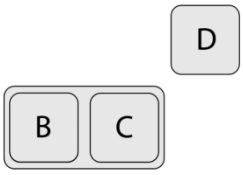


图6.13　BC组合

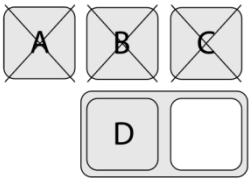


图6.14　从D开始的情况

综上可总结出，第一门课程有 4 种选项，第二门课程的选项不包括第一门课程以及第一门课程之前的课程。

有了这个思路，让我们看一下不用回溯写出来的代码是什么样的。

```
01 class Solution():
02    def solveCombinationBadWay(self,array,n):          #array 是课程列表，n 代表可选数量
03        solution = []                                   #solution 是组合列表
04        for i in range(len(array)):                     #第一门课的选择
```

```
05          newSolution1 = solution + [array[i]]                    #把课程加入新组合列表
06          newArray = array[i + 1:]                                #把课程从新课程列表剔除
07          for i in range(len(newArray)):                          #第二门课的选择
08              newSolution2 = newSolution1 + [newArray[i]]         #把课程加入组合列表
09              print (newSolution2)                                #输出组合列表
```

注意，我们没有把课程加入原本的组合列表，而是每一次都创建一个新的组合列表。这样做是因为每一次加的课程都是不同的。加入第一门课时，创建了组合列表[A]，[B]，[C]，[D]，加入第二门课时，创建了组合列表[A，B]，[A，C]，[A，D]，[B，C]，[B，D]，[C，D]。

另外注意，我们加入第一门课后更新了课程列表：

```
06  newArray = array[i + 1:]                    #更新课程列表
```

也就是说，第二门课的选择是已选的第一门课之后的课程。比如，第一门课选了 B，更新后的课程列表是[C，D]。

这是不用回溯的写法，如果 *n* 的值很小，这样写没什么问题，前提是你要知道 *n* 是多少。但是，一旦 *n* 变大，代码就会变得烦琐，所以我们应该利用回溯来改善以上代码。

6.3.2　最终代码

定义一个 helper 方法帮我们概括上面重复的代码。以下是 helper 方法的完整代码。你可以先看一看它和上面的代码有什么区别，下面我们会详细解释。

```
01   def helper(self,array,n,solution):
02       if len(solution) == n:                          #如果已选 n 门课程
03           print (solution)                            #输出结果
04           return                                      #返回上一级
05       for i in range(len(array)):                     #遍历课程列表
06           newSolution = solution + [array[i]]         #新建组合列表
07           newArray = array[i + 1:]                    #新建课程列表
08           self.helper(newArray,n,newSolution)         #回溯
```

首先我们来看一下 helper 方法的参数。

- array，课程列表，存储可选课程。如果第一次调用时 array=[A,B,C,D]，那么第二次调用时 array 可能等于[B,C,D]、[C,D]或者[D]。
- n，选课数量。在这 4 门课中选择 2 门的话，n=2。n 是一个常量，不会改变。
- solution，当前的组合列表。solution 一开始为空，随着每一门课程的加入慢慢增长。

接下来看一下 helper 方法履行的任务。

1. 向组合列表中加入一门课程。

```
01    def helper(self,array,n,solution):
02        for i in range(len(array)):                    #array 中的科目选项
03            newSolution = solution + [array[i]]        #加入组合列表
```

array 中是我们剩余的可选科目，solution 是我们现有的、未满的组合列表。依次把 array 中的可选科目加入新创建的组合列表中。我们没有继续使用原有的组合列表，因为每一次加入的课程不同。

2. 更新课程列表。

```
04     newArray = array[i + 1:]                        #创建新课程列表，更新列表
```

newArray 是下一门课可选的课程。我们新建了一个列表，而不是更新原有的列表，因为每一次剩下的可选课程都不同。

3. 继续选择剩余课程。

```
05     self.helper(newArray,n,newSolution)            #调用 helper 方法选择剩余课程
```

有了 newSolution 与 newArray 后就可以调用 helper 方法继续选择剩余的课程了。

4. 检查是否已选择 n 门课程。

```
01    def helper(self,array,n,solution):
02        if len(solution) == n:                      #如果 solution 中有 n 门课程
03            print (solution)                        #输出结果
04            return                                  #返回
```

如果不设定一个结束条件的话，helper 方法不知道什么时候返回结果。所以，在一开始就要明确，如果 solution 中已经有了 n 门课程，就输出组合列表。输出后，返回上一次被调用的地方，继续没有走完的 for 循环，输出其他组合。

有了 helper 方法，只需要一开始的时候调用一次它就可以了。

```
01   def solveCombination(self,array,n):
02       self.helper(array,n,[])              #调用 helper 方法
```

注意：一开始传入的 solution 是一个空的列表，因为还没有开始选课。

其实，和不用回溯的代码对照一下你会发现两者没有本质上的区别。不用回溯就是把 helper 方法展开，那样写会十分麻烦但逻辑上和回溯的代码没有差别。

来看一下完整的代码，如代码 6.3 所示。

代码 6.3　用回溯算法解决组合问题

```
01 class Solution():
02   def solveCombination(self,array,n):
03      self.helper(array,n,[])                          #调用 helper 方法
04   def helper(self,array,n,solution):                  #helper 方法
05      if len(solution) == n:                           #如果 solution 中有 n 门课程
06         print (solution)                              #输出结果
07         return                                        #返回被调用的地方
08      for i in range(len(array)):                      #遍历每一个科目选项
09         newSolution = solution + [array[i]]           #把课程加入新的组合列表
10         newArray = array[i + 1:]                      #创建新课程列表，更新列表
11         self.helper(newArray,n,newSolution)           #调用 helper 方法选择剩余课程
```

来试试解决小朗朗的问题：

```
Solution().solveCombination(['A','B','C','D'],2)
```

下面是输出结果：

```
['A', 'B']
['A', 'C']
['A', 'D']
['B', 'C']
['B', 'D']
['C', 'D']
```

6.4　查找单词问题

你玩过报纸上那种找单词的游戏吗？就是那种在一堆字母中横向或竖向找出单词的游戏。小朗朗现在在玩一个和那个很像的游戏，只不过现在不但可以上下左右斜着连接字母，还可以转弯。看一下图 6.15，这是游戏的盘面，图 6.16 展示的是遵循规则找出来的 4 个单词：“world”“week”“rose”“reef”。

a	c	r	y	l
l	w	o	r	i
a	f	d	l	c
k	e	e	w	e
o	d	r	o	s

图6.15　游戏盘面

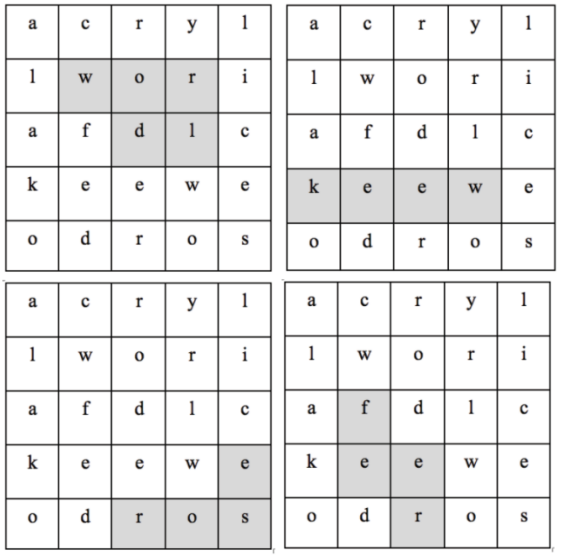


图6.16　找出的单词

给出一个单词，我们的目标是通过回溯算法得出这个单词是否存在于盘面中。比如，给予“world”就返回 True，给予“hello”就返回 False。

6.4.1　问题求解

下面我们手动模拟一下算法，通过这个过程能够找到写代码的逻辑。以“week”单词为例。

首先，需要看一下盘面中的 25 个字母中有没有“w”。如图 6.17 所示，找到了一个“w”。接下来，第二个字母是“e”，来看一下“w”的上下左右有没有“e”。因为没有找到“e”，所以只好寻找另一个“w”。

如图 6.18 所示，找到了第二个“w”之后，发现它的左边和右边都是“e”。

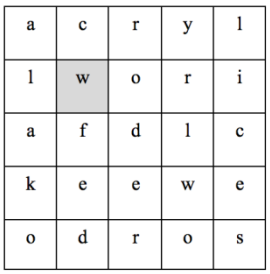


图6.17　找到第一个字母“w”

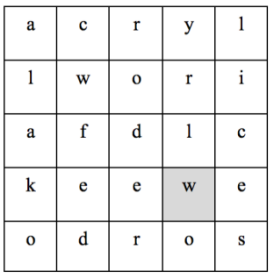


图6.18　重新找“w”

遵守上下左右的顺序，选择左边的“e”，如图 6.19 所示。

接下来的步骤就不一一展示了，到最后，成功地找出了“week”这个单词，如图 6.20 所示，所以返回 True。

a	c	r	y	l
l	w	o	r	i
a	f	d	l	c
k	e	e	w	e
o	d	r	o	s

图6.19　第二个字母“e”

a	c	r	y	l
l	w	o	r	i
a	f	d	l	c
k	e	e	w	e
o	d	r	o	s

图6.20　完整单词

下面我们来看一下不用回溯写出的代码。当你不清楚怎么用回溯写出简短的代码时，可以尝试先用“笨代码”把逻辑写出来。这样能够看清楚哪段代码是重复的，哪个函数是需要传入的，哪个条件是输出条件，看清楚之后写回溯方法就很容易了。

```
01 def wordSearchBadWay(self,board,word):              #传入盘面，单词
02       for i in range (len(board)):
03         for j in range (len(board[0])):             #从任何一个字母开始
04               if board[i][j] == word[0]:            #如果 board[i][j]是单词的第 1 个字母
05                  board[i][j]=""                     #把字母改成""，以免二次利用
06                  if board[i-1][j] == word[1]:       #如果[i][j]的上面是单词的第 2 个字母
07                     board[i-1][j]=""                #空盘面字母
08                     if board[i-2][j] == word[1]:        #上
09                        ...                          #看[i][j+2]的上下左右有没有第 3 个字母
10                        (return True)                #如果最后找到单词，输出 True
11                     if board[i][j] == word[1]: #下
12                        ...
13                        (return True)
14                     if board[i-1][j-1] == word[1]:      #左
15                        ...
16                        (return True)
17                     if board[i-1][j+1] == word[1]:      #右
18                        ...
19                        (return True)
20                     board[i-1][j]=word[1]          #上下左右都不行，改回盘面原来的字母
21                  if board[i+1][j] == word[1]:       #如果[i][j]下面是单词的第 2 个字母
22                     ...                            #看[i][j-1]的上下左右有没有第 3 个字母
23                  if board[i][j-1] == word[1]:       #如果[i][j]左面是单词的第 2 个字母
24                     ...
```

```
                    if board[i][j+1] == word[1]:   #如果[i][j]右面是单词的第 2 个字母
                        ...
                    board[i][j]=word[0]
                                #上下左右一圈下来没有找到单词，改回字母，从另外的地方开始
    return False                                    #没有找到单词，返回 False
```

上面的代码看上去非常乱，但它的逻辑其实很好理解。它的基本逻辑是：从盘面上任何字母（1）开始，如果（1）是单词的首字母，看它的上下左右有没有单词的第二个字母，如果有，称这个字母为（2），再看（2）的上下左右有没有单词的第三个字母……就这样一直检查下去；如果没有，就改变（1），换一个字母开始。

需要注意的是，当我们找到了一个对应的字母时，需要把它从盘面中删除（改成空字符串），这样在下面的检查过程中不会再次使用该字母。检查后，如果该字母的上下左右没有对应的剩余单词，需要把它再次填入盘面。

看一下图 6.21，是不是一下感觉代码整齐了许多？每一个黑框中的代码都是重复的。仔细观察，你会发现每一个黑框都包着 4 个小黑框。

```
01 def wordSearchBadWay(self,board,word):          #传入盘面，单词
02         for i in range (len(board)):
03             for j in range (len(board[0])):      #从任何一个字母开始
04                 if board[i][j] == word[0]:       #如果 board[i][j]是单词的第 1 个字母
05                     board[i][j]=""               #把字母改成""，以免二次利用
06                     if board[i-1][j] == word[1]:  #如果[i][j]的上面是单词的第 2 个字母
07                         board[i-1][j]=""          #空盘面字母
08                         if board[i-2][j] == word[1]:  #上
09                             ...                  #看[i][j+2]的上下左右有没有第 3 个字母
10                             (return True)        #如果最后找到单词，输出 True
11                         if board[i][j] == word[1]:  #下
12                             ...
13                             (return True)
14                         if board[i-1][j-1] == word[1]:  #左
15                             ...
16                             (return True)
17                         if board[i-1][j+1] == word[1]:  #右
18                             ...
19                             (return True)
20                         board[i-1][j]=word[1]    #上下左右都不行，改回盘面字母
21                     if board[i+1][j] == word[1]:  #如果[i][j]下面是单词的第 2 个字母
22                         ...                      #看[i][j-1]的上下左右有没有第 3 个字母
23                     if board[i][j-1] == word[1]:  #如果[i][j]左面是单词的第 2 个字母
24                         ...
25                     if board[i][j+1] == word[1]:  #如果[i][j]右面是单词的第 2 个字母
26                         ...
27                     board[i][j]=word[0]
28                         #上下左右一圈下来没有找到单词，改回字母，从另外的地方开始
29         return False                             #没有找到单词，返回 False
```

图6.21　重复代码

注意：我们省略了一些前提条件，比如，如果 i 小于 0 则不检查。

写这样的代码的前提条件是已知单词的长度，不然我们无法得知在第几环输出结果。

6.4.2　最终代码

现在清楚地看到了重复的代码结构，就可以轻松地写出回溯算法了。定义一个 helper 回溯方法概括重复的代码。下面我们会进行解释。

```
01   def helper(self,board,current,row,column):
02       if len(current)==0:                              #如果找到单词
03           return True                                  #返回 True
04       if row>=0 and row<len(board) and column>=0 and column<len(board[0]):
05           if board[row][column] == current[0]:         #如果有字母对应剩余单词首字母
06               board[row][column]=""                    #把该字母从盘面中去除
07               if self.helper(board,current[1:],row-1,column):    #检查第 2 个字母，上
08                   return True
09               if self.helper(board,current[1:],row+1,column):    #下
10                   return True
11               if self.helper(board,current[1:],row,column-1):    #左
12                   return True
13               if self.helper(board,current[1:],row,column+1):    #右
14                   return True
15               board[row][column]=current[0]            #把字母填回到盘面
16        return False                                    #如果上下左右都没有剩余字母，返回 False
```

上面是 helper 方法的完整代码，也就是“笨代码”中黑框里的代码。只不过我们把黑框中的小黑框改成了调用 helper 方法。

来看一下 helper 方法的参数：

- board，游戏的盘面，一个由字母组成的二维列表。不一定是 n*n，可能是 6*7、8*8、9*1……
- current，剩余的单词，比如“week”“eek”“ek”“k”“”。
- row，当前字母在盘面的横坐标。
- column，当前字母在盘面的纵坐标。

来看一下 helper 方法履行的任务：

1. 检查当前盘面坐标对应的字母是不是剩余单词的首字母。

```
01   def helper(self,board,current,row,column):
02       if row>=0 and row<len(board) and column>=0 and column<len(board[0]):
```

```
                                                    #前提条件
03          if board[row][column] == current[0]:    #如果字母对应剩余单词的首字母
04              board[row][column]=""               #把字母从盘面中去除
```

如果当前坐标不在盘面范围内（如果坐标小于 0 或大于盘面长度），跳过该坐标。如果符合前提条件，并且两个字母相同，把字母从盘面中去除。

2. 如果对应，检查坐标的上下左右是否对应剩余单词的首字母。

```
05              if self.helper(board,current[1:],row-1,column): #上
06                  return True
07              if self.helper(board,current[1:],row+1,column): #下
08                  return True
09              if self.helper(board,current[1:],row,column-1): #左
10                  return True
11              if self.helper(board,current[1:],row,column+1): #右
12                  return True
13              board[row][column]=current[0] #把字母填回到盘面
14          return False                      #如果上下左右都没有剩余字母，返回 False
```

调用 helper 方法返回的布尔值代表的是：以当前坐标为起点是否能够找出剩余的单词（current[1:]）。我们有 4 个选择，所以要调用 helper 方法 4 次。如果没有返回 True，走到了第 13 行代码，那么就把字母填回到盘面上。

3. 检查是否已经找到单词。

```
01    def helper(self,board,current,row,column):
02        if len(current)==0:                   #如果找到单词
03            return True                       #返回 True
```

在一开始的时候，如果剩余单词是一个空字符串，我们就没有往下走的意义了，直接返回 True 就可以了。

有了 helper 方法，需要遍历盘面中的字母并调用它解决问题。

```
01    def wordSearch(self, board, word):
02        for i in range (len(board)):
03            for j in range (len(board[0])):           #遍历盘面
04                  if self.helper(board,word,i,j):     #调用 helper 方法
05                      return True
06        return False
```

完整代码如代码 6.4 所示。

代码 6.4　用回溯算法解决查找单词问题

```
class Solution():
    def wordSearch(self, board, word):
        for i in range (len(board)):
            for j in range (len(board[0])):                    #遍历盘面
                if self.helper(board,word,i,j):                #调用 helper 方法
                    return True
        return False
    def helper(self,board,current,row,column):                 #helper 方法（回溯方法）
        if len(current)==0:                                    #结束条件
            return True
        if row>=0 and row<len(board) and column>=0 and column<len(board[0]): #前提条件
            if board[row][column] == current[0]:               #如果两个字母对应
                board[row][column]=""
                if self.helper(board,current[1:],row-1,column):    #上下左右检查剩余字母
                    return True
                if self.helper(board,current[1:],row+1,column):
                    return True
                if self.helper(board,current[1:],row,column-1):
                    return True
                if self.helper(board,current[1:],row,column+1):
                    return True
                board[row][column]=current[0]
        return False
```

那我们就尝试一下查找“week”这个单词：

```
board = [["a", "c", "r", "y", "l"],["l", "w", "o", "r", "i"],["a", "f", "d", "l", "c"],["k",
"e", "e", "w", "e"],[ "o", "d", "r", "o", "s"]]
Solution().wordSearch(board, "week")
```

返回：

```
True
```

6.5　八皇后问题

保安部负责人小安面临着一个难题，他需要在一个 8 × 8 公里的区域里修建 8 个保安站点，并确保每一行、每一列和每一条斜线上都只有一个保安站点。苦恼的小安尝试着布置了很多遍，但每一次都不符合要求。小安只好求助编程部的小朗朗，没过多久，小朗朗就把好几个布置方案（实际上，这个问题有 92 种答案）发给了小安，其中包括图 6.22 与图 6.23 所示的这两个示

例。摆脱了难题的小安想知道小朗朗是怎么做到的。

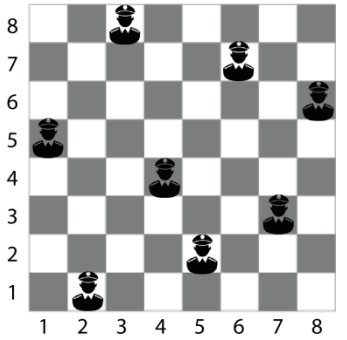


图6.22　布置图样一

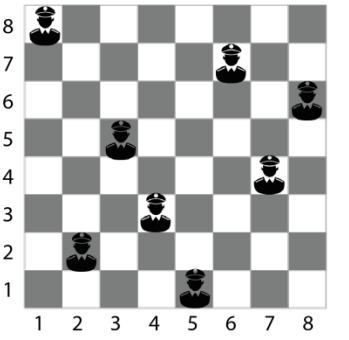


图6.23　布置图样二

小安的问题其实就是八皇后问题。在国际象棋中，皇后能够上下左右斜无束缚地进行攻击。八皇后问题的原题是如何在一个 8×8 的棋盘中放置 8 个皇后，并令她们互相攻击不到对方。八皇后问题是回溯算法的代表性问题之一。希望通过这个问题，可以让你更好地领会回溯算法的思想。

6.5.1　问题求解

首先，我们分析一下这个问题。问题中说到每一行、每一列和每一条斜线上都必须有且只能有一个保安站。它告诉了我们第一行的 8 个格子中肯定有一名保安，可以从这里入手。

那么，利用“每一行都必须有一个保安”的前提条件，假设第 1 行有一个保安 A。我们首先假设保安 A 在(1, 1)格，如图 6.24 所示，然后再判断这个假设是否成立。

然而我们并没有办法直接判断保安 A 是否能够留在(1, 1)格，因为其他几个保安还没有被安置，如图 6.25 所示。

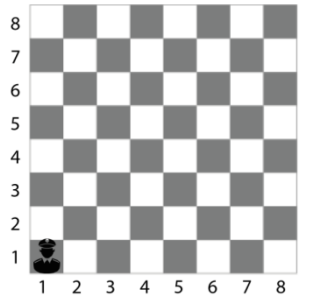


图6.24　假设保安A在(1, 1)格

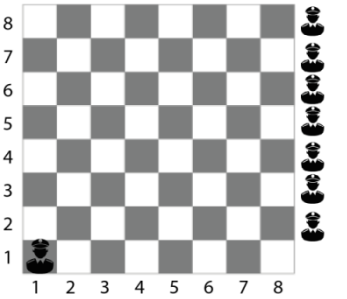


图6.25　未被安置的7名保安

所以只能继续假设下去。假设第 2 行有一个保安 B，他在第 2 行的(2, 3)格；假设第 3 行有一个保安 C，他在第 3 行的(3, 5)格……就这样一直假设下去，如图 6.26 所示。

这个假设的过程有两种可能的结果。

第一种结果：我们成功地假设到了第 8 行，也就是说，找到了正确答案之一。

第二种结果：在还没到第 8 行的时候就走不下去了，也就是说，陷入了死局，如图 6.27 所示。

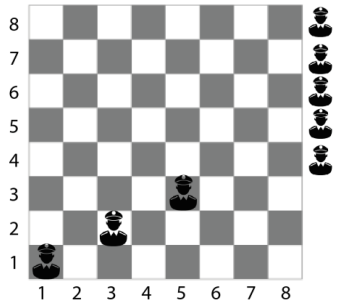


图6.26　继续假设剩余保安的位置

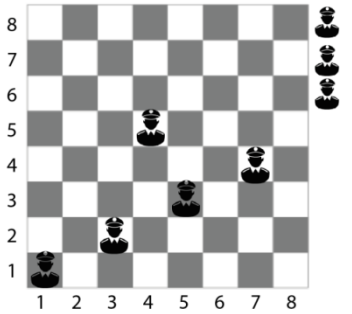


图6.27　陷入死局的情况示例

如果是第一种可能，那就太好了，我们直接把结果输出。但如果是第二种可能，我们则需要退一步，重新假设前一行保安的位置。在必要的情况下，我们会一直退回到第一行，重新假设保安 A 的位置。

总结一下思路，如图 6.28 所示。以第一行为起点，假设保安位置，目标是一直成功地假设出有效的第 8 行保安的位置。如果中途陷入死局，则改变上一行保安的位置。

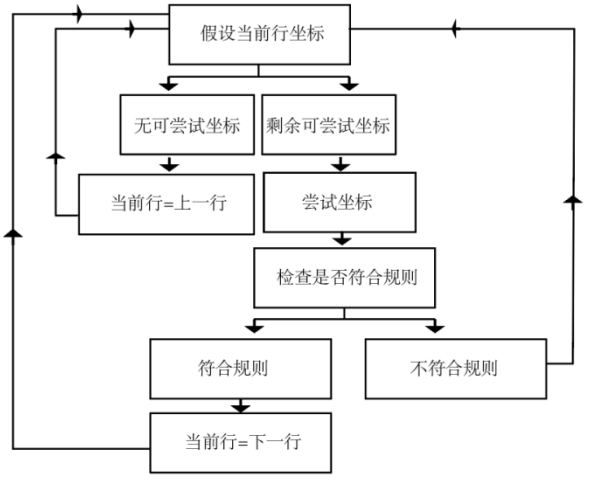


图6.28　解题思路图

现在尝试不用回溯的方法把思路写出来。

```
for column in range(0,8)
     place 保安A in (0, column)      #假设保安A在第一行的某个格子中
     if valid:                       #如果把保安放置在这个格子和其他的安排不冲突则继续
          for column in range(0,8)
               place 保安B in (1, column) #假设保安B在第二行的某个格子中
               if valid                   #如果不冲突则继续
                    for column in range(0,8)
                    place 保安C in (2, column)      #就这样一直到第8行
                    if valid
                              for column in range(0,8)
                              place 保安D in (3, column)
                              if valid
                                        ….
```

用笨代码也能得到正确的 92 种答案，但为什么希望用回溯算法写呢？首先，笨代码违背代码应有的简单性。再有，笨代码只适用于一个 8×8 的区域，如果小安下次需要安排一个 10×10 的区域就不合适了。我们希望代码能够用于不同大小的区域。

其实可以发现上面的代码很多地方都是重复的。如图 6.29 所示，代码一环套一环，每一环的结构都是一样的。每一次做完一行保安位置的假设后，都在继续做剩余行保安位置的假设。

那么是不是可以构造一个方法，其功能就是概括这些重复的部分呢？那样，只要传进不同的参数就可以重复使用这个方法了。

```
for column in range(0,8)
     place 保安 A in (0, column)
     if valid:
          for column in range(0,8)
               place 保安 B in (1, column)
               if valid
                    for column in range(0,8)
                         place 保安 C in (2, column)
                         if valid
                              for column in range(0,8)
                              place 保安 D in (3, column)
                                   if valid
                                        ....
```

图6.29　重复的环形结构

6.5.2　最终代码

上面说到了创建一个回溯方法来概括“假设当前行的保安位置→检查是否成立→假设剩余

行的保安位置”这一过程，我们称这个方法为 helper 方法。来看一下 helper 方法的完整代码，后面我们会详细解释。

```
01    def helper(self, columnPositions, rowIndex, n):    #helper 方法
02        if rowIndex == n:                              #rowIndex 代表当前行
03            self.printSolution(columnPositions,n)      #如果走完了最后一行就输出结果
04            return                                     #返回上一级
05        for column in range(n):
06            columnPositions[rowIndex] = column         #假设第 rowIndex 行的保安位置
07            if self.isValid(columnPositions, rowIndex):      #如果可行
08              self.helper(columnPositions, rowIndex+1, n)    #继续假设剩余行的保安位置
```

首先，来看一下 helper 方法的参数：columnPositions、rowIndex 和 n。

- columnPositions，用于存储保安位置的数组。columnPositions 的坐标代表第几行，存储值代表第几列。
- rowIndex，记录当前行数。从 rowIndex = 0 开始到 rowIndex = 8 时结束。
- n，区域的大小。这里的 n 为 8。

现在，来看一下 helper 方法履行的任务：

1. 假设当前行保安可在的不同位置：

```
01    def helper(self, columnPositions, rowIndex, n):
02        for column in range（n）:              #有 n 个位置选择
03            columnPosition[rowIndex] = column  #假设保安在（rowIndex，column）坐标
```

假设当前保安在(rowIndex, column)位置，更新 columnPositions。

注意：写代码时，保安的坐标值需要减 1，因为 Python 的列表索引从 0 开始。如果当前行是第 1 行，rowIndex 则为 0。

2. 判断假设的保安位置是否合理。

```
04    if self.isValid(columnPositions, rowIndex): #检查假设是否成立
```

需要创建一个 isValid 方法来判断当前的假设是否合理。比如，第 1 行的保安 A 在(1, 1)，假设第 2 行保安 B 的位置在(2, 1)，两名保安被放置在同一列，isValid 方法会返回 False，所以也就不会浪费时间去假设第 3 行的保安 C 的位置了，而是重新假设保安 B 的位置。

注意：如果 n 个坐标都不符合规则，程序会自动跳回到上一次调用 helper 方法的地方。它会继续没走完的 for 循环，重新假设上一行的保安位置。

3. 如果合理，继续假设下面几行的保安位置。

```
05    self.helper(columnPositions, rowIndex+1, n) #继续假设
```

需要重新调用 helper 方法，传入下一行的参数。比如，假设完第一行保安 A 在(1, 1)格，并且假设成立，则继续调用 helper 方法，传入更新后的 columnPositions 与 rowIndex+1，假设第 2 行到第 8 行的保安位置。

4. 判断 8 名保安是否都已被安置成功。

```
01 def helper(self, columnPositions, rowIndex, n):
02  if rowIndex == n:                                  #如果 8 名保安都被安置
03    self.printSolution(columnPositions,n)            #输出结果
04   return                                            #返回上一行的假设
```

如果 8 名保安都成功地被安置在了这个 8×8 的区域，则输出结果。如果目的是输出一个结果，便可在此终止程序。如果需要输出所有的结果，则在输出完这个结果后返回到第 7 行，重新假设第 7 名保安的位置。

设计完 helper 方法，只需要调用它一次就可以解决小安的问题了。

```
01   def solveNQueens(self, n):
02      self.helper([-1]*n, 0, n)
```

注意：因为我们还没有开始布置保安位置，所以一开始传入的 columnPositions 是一个长度为 n，且所有值都为-1 的数组。

以下代码是用笨方法写出来的死代码，适用于 4×4 的棋盘。其实仔细看一看，笨的代码就是把 helper 方法展开了，输出的结果和使用 helper 方法输出的结果完全一致。

```
01  def solveNQueensBadWay(self):
02      columnPositions = [-1]*4       #columnPositions 是用于存储 4 名保安位置的数组
03      for i in range(4):
04          columnPositions[0] = i                          #假设第 1 行的保安位置
05          if self.isValid(columnPositions, 0):            #如果成立
06          for i in range(4):
07              columnPositions[1] = i                      #假设第 2 行的保安位置
08              if self.isValid(columnPositions, 1):        #如果成立
09                  for i in range(4):
10                      columnPositions[2] = i              #假设第 3 行的保安位置
11                      if self.isValid(columnPositions, 2):    #如果成立
12                          for i in range(4):
13                              columnPositions[3] = i      #假设第 4 行的保安位置
```

```
14                                  if self.isValid(columnPositions, 3):       #如果成立
15                                      self.printSolution(columnPositions,4) #输出结果
```

代码 6.5 所示的就是小多多为了解决小安的问题写出的完整代码。小多多考虑得很周到，她想小安可能在其他城市安排保安站时还会面临相似的问题，所以她的代码适用于任何 n×n 的区域。

代码 6.5　用回溯算法解决八皇后问题

```
01 class Solution(object):
02
03    def solveNQueens(self, n):
04       self.helper([-1]*n, 0, n)
05
06    def helper(self, columnPositions, rowIndex, n):    #helper 方法
07       if rowIndex == n:                               #rowIndex 代表当前行
08          self.printSolution(columnPositions,n)        #如果走完了所有行就输出结果
09          return                                       #返回到上一行的假设
10       for column in range(n):
11          columnPositions[rowIndex] = column           #假设第 rowIndex 行的保安位置
12          if self.isValid(columnPositions, rowIndex): #如果可行
13            self.helper(columnPositions, rowIndex+1, n)    #继续假设剩余行的保安位置
14
15    def isValid(self, columnPositions, rowIndex):      #isValid 方法检查位置是否合理
16       for i in range(rowIndex):
17          if columnPositions[i] == columnPositions[rowIndex]:  #检查同列是否有保安
18             return False
19          elif abs(columnPositions[i]-columnPositions[rowIndex]) == rowIndex-i:
20             return False                                #检查两条斜线上是否有保安
21       return True
22
23    def printSolution(self, columnPositions, n):       #printSolution 方法输出结果
24       for row in range (n):
25         line = ""
26         for column in range (n):
27           if columnPositions[row] == column:
28             line += "Q "
29           else:
30             line += ". "
31         print(line)
32       print("\n")
33
34 Solution().solveNQueens(8)                            #调用 solveNQueens 解决 8 皇后问题
```

图 6.30 至图 6.32 所示的是得出的 92 个结果的前 3 个。

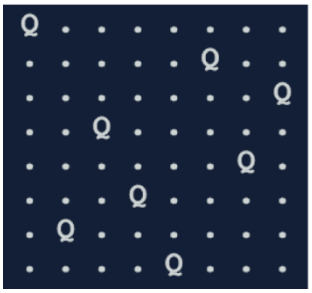


图6.30　输出结果1

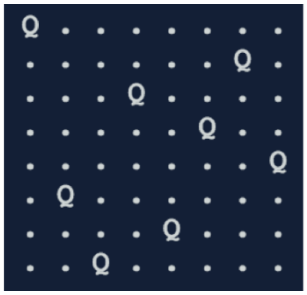


图6.31　输出结果2

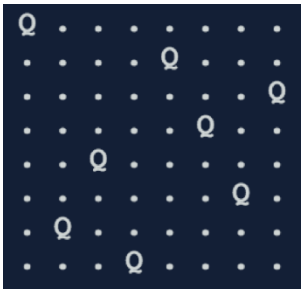


图6.32　输出结果3

6.6　教你解数独

小朗朗最近对数独着了迷。在玩游戏的时候，计算机程序能够立刻得出答案，好奇的他想知道计算机是怎么得出结果的。数独的游戏规则是根据盘面上的已知数字推理出空格里的数字，要求每一行、每一列、每一个粗线格中均含数字 1~9，且不重复，如图 6.33 所示。

	1				7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.33　数独游戏

6.6.1　问题求解

面对这样的问题，没有像公式一样的直接解决方式，所以只能靠尝试。如图 6.34 所示，就像在走迷宫，没有地图，所以面对每一个岔路口都只能以探索的形式深入，一旦发现路线不对就返回岔路口，选择另一个分支。

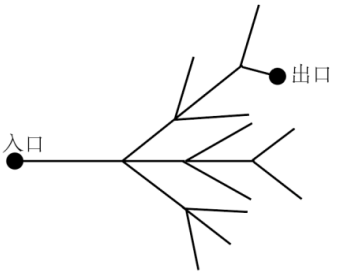


图6.34　迷宫

把这个比喻用在数独上的话，每一个空格就是一个岔路口，而每一个岔路口都有 9 种选择。我们肯定末端有一个出口。那么，从第 1 个岔路口出发。如图 6.35 至图 6.37 所示，假设第 1 个空格中的数字：1 不可以，2 也不可以，3 可以。

1	1				7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.35　假设第一个空格的数字为1

2	1				7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.36　假设第一个空格的数字为2

接下来假设第 2 个空格，1、2、3 都不可以，4 可以，如图 6.38 所示。

3	1				7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.37　假设第1个空格的数字为3

3	1	4			7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.38　假设第2个空格的数字

接下来假设第 3 个空格：1 不可以，2 可以。接着假设第 4 个空格：糟糕，1~9 都不可以。看来是陷入了死局，如图 6.39 所示。

怎么办？只能退回一步。如图 6.40 所示，重新假设第 3 个空格中的数字：1~5 都不可以，6 可以。

3	1	4	2	?	7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.39　第4个空格没有选择

3	1	4	6		7	8	5	
8		6						
	2			9		6		
							1	2
2			5	8			7	
						5		
			7		2			
		8		1				
	7			6				

图6.40　改变第3个格中的数字

就这样一直假设下去，如果盘面上的数字相互矛盾则改变上一格的数字，再次尝试。当我们到达最后一个空格并且成功填入数字时，即可输出答案。

注意：输出答案的前提是题目有解。如果题目有两个或更多答案，程序可输出所有答案。

其实这个问题和八皇后问题本质上没有什么区别，这两个问题都是循环“假设—检查—假设”这一过程。图 6.41 总结了我们的解题思路。

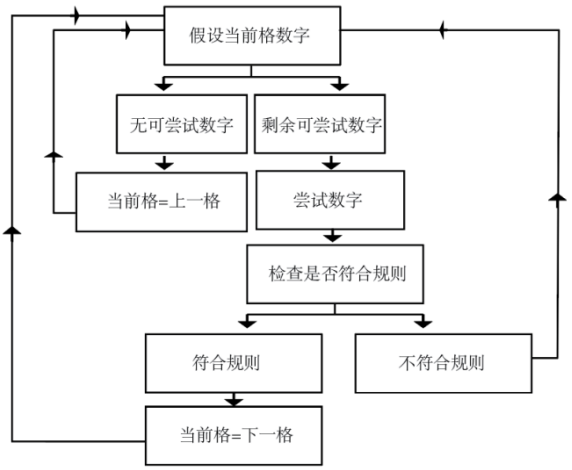


图6.41　数独解题思路图

和八皇后问题一样，我们来看一下如果不用回溯写出来的代码是什么样的：

```
01 def solveSudoku(self):
02   if 格子 1 为空:                          #如果第 1 格为空
03     for num in range(10):                 #依次尝试 9 个数字
04       if valid(格子 1=num):                #如果数字成立
05            格子 1=num                      #把数字填入格中
06          if 格子 2 为空:                   #继续第 2 格
07               for num in range(10):       #依次尝试 9 个数字
08                 if valid(格子 2=num):     #如果数字成立
09                      格子 2=num           #把数字填入格中
10                       格子 3-81...        #继续第 3 格至第 81 格
11                      格子 2=0             #刷新第 2 格，重设 0
12         else:                             #如果第 2 格有已知数
13               if 格子 3 为空:             #跳过第 2 格，继续第 3 格
14                  ...                      #剩余代码依次类推
15            格子 1=0                        #刷新第 1 格，重设 0
16   else:                                   #如果第 1 格有已知数
17      if 格子 2 为空:                      #跳过第 1 格，继续第 2 格
18         ...
```

注意：每一次假设后都要把格子还原回 0，以便下一次假设时盘面上的数字不冲突。如果不归零，那么可能陷入死局后退回到第 1 个空格时，第 2 个空格里已经有了 9，这会影响我们的判断。

不用回溯的话，需要把 81 个格子的假设过程全部写出来。上面的代码只是写了前两个格子

的代码，可想要写到第 81 格代码会有多长。虽然这样也能返回正确答案，但这样写太费事，并且很累赘。

6.6.2 最终代码

观察图 6.42，你会发现每一个格子都是在重复上一个格子的代码，就像俄罗斯套娃，娃娃们除了格子坐标不同之外其他完全一致。

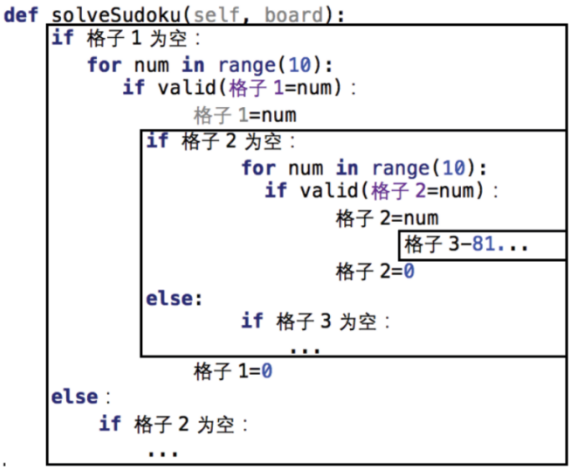


图6.42　重复的代码

利用回溯算法，可以把这些重复的代码概括一下，这样当传入不同的参数时就可以避免重复。下面我们定义一个 helper 方法来帮助概括重复的代码，以下是 helper 方法的完整代码，下面会详细解释。

```
01   def helper(self, board, index):
02       if index >= 81:                              #如果盘面已满
03           self.printSolution(board)                #输出结果
04           return                                   #返回上一级
05       if board[index] == 0:                        #如果格子为空
06           for i in range(1, 10):
07               if self.isValid(board, index, i):    #如果假设数字成立
08                   board[index] = i                 #设置数字
09                   self.helper(board, index + 1)    #假设其他空格
10                   board[index] = 0                 #改回空格
11       else: self.helper(board,index+1)             #如果格子不为空，跳过
```

首先，来看一下 helper 方法的两个参数：

- index，当前空格的坐标。index = 0 时在第 1 格，index = 80 时在最后一格。
- board，存储盘面数字的数组，由数字 0~9 组成，0 代表空格。数组的坐标依行排序，例如，第 1 行第 1 格的坐标为 0，第 3 行第 1 格的坐标为 18。

看一下 helper 方法履行的几个任务：

1. 如果当前格为空格，假设数字。

```
01 def helper(self, board, index):
02       if board[index] == 0:       #如果当前格为空格
03          for i in range(1, 10):   #依次假设 1~9
```

if 语句建立假设前提，记住我们用 0 代表空格。

2. 验证假设是否成立。

```
04       if self.isValid(board, index, i):    #如果假设成立
05          board[index] = i                  #填入数字
```

需要建立一个 isValid 方法。当传入当前盘面、当前格坐标和当前格的假设数字时，如果假设数字与盘面上的其他数字不冲突，方法会返回 True。只要假设成立，我们就在盘面上填入数字。

注意：如果数字 1~9 都不符合规则，程序会自动跳回到上一次调用 helper 方法的地方。它会继续上一格没走完的 for 循环，重新假设上一格的数字。

3. 如果成立，继续假设剩余盘面。

```
06       self.helper(board, index + 1)    #继续假设
07       board[index] = 0
```

需要再次调用 helper 方法，传入不同的参数。比如，假设第 1 个空格中的数字为 2，并且数字成立，那么调用 helper(board,1)就代表继续从第 2 格假设到最后一格。第 2 格假设完后，调用 helper(board,2)就代表继续从第 3 格假设到最后一格。

我们调用 helper 方法后跳了回来，走到了第 7 行代码，就证明当前假设的数字 i 不成立，或者已经输出了结果，这时需要把格子改回空格，以便继续假设其他数字。

4. 检查盘面是否已满。

```
01 def helper(self, board, index):
02    if index >= 81:                          #如果盘面已满
03          self.printSolution(board)          #输出结果
04          return                             #返回上一格
```

在假设当前格中数字之前要先检查一下我们是否已经得出答案。如果 index=81 就代表 81 个格子中都填上了成立的数字。那么，只需要输出答案就可以了。

注意：返回上一行的目的是输出可能的其他答案，因为数独有时答案不唯一。

现在解决数独问题只需要调用一次 helper 方法就可以了。

```
01  def solveSudoku(self, board):
02      self.helper(board, 0)          #index = 0,从第 1 格开始
```

下面让我们来看一下完整的代码，如代码 6.6 所示。

代码 6.6　用回溯算法解决数独问题

```
01 class Solution():
02   def solveSudoku(self, board):
03      self.helper(board, 0)
04
05    def helper(self, board, index):
06      if index >= 81:                      #如果盘面已满
07         self.printSolution(board)         #输出结果
08         return                            #返回上一格
09     if board[index] == 0:                 #如果当前格为空格
10         for i in range(1, 10):            #依次假设 1~9
11            if self.isValid(board, index, i): #如果假设成立
12               board[index] = i           #填入数字
13               self.helper(board, index + 1) #继续假设
14               board[index] = 0
15      else:                                #如果当前格有已知数
16          self.helper(board,index+1)       #跳过此格
17
18   def printSolution(self, board):         #输出结果
19      for i in range(0,81,9):
20         print(board[i:i+9])
21
22   def isValid(self, board, index, num):  #isValid 方法检查当前假设是否合理
23      row = index // 9                     #当前格子的行数
24      column = index % 9                   #当前格子的列数
25      for i in range(index + 9, 81, 9):    #检查和同列（下方）的格子是否矛盾
26         if board[i] == num:
27            return False
28      for i in range(index - 9, -1, -9): #检查和同列（上方）的格子是否矛盾
29         if board[i] == num:
30            return False
```

```
31        for i in range(9 * row, 9 * (row + 1)): #检查和同行的格子是否矛盾
32            if board[i] == num:
33                return False
34        for i in range(row - row % 3, 3 + row - row % 3):
                                                  #检查和同粗线的格子是否矛盾
35            for j in range(column - column % 3, 3 + column - column % 3):
36                if board[j + i * 9] == num:
37                    return False
38        return True
```

让我们解决数独问题：

```
Solution().solveSudoku(
    [4, 1, 0, 0, 0, 7, 8, 5, 0,
     8, 0, 6, 0, 0, 0, 0, 0, 9,
     0, 2, 0, 0, 9, 0, 6, 0, 0,
     0, 0, 4, 0, 0, 0, 0, 1, 2,
     2, 0, 0, 5, 8, 0, 0, 7, 0,
     0, 0, 0, 0, 0, 0, 5, 0, 0,
     0, 0, 0, 7, 0, 2, 0, 0, 0,
     0, 0, 8, 0, 1, 0, 0, 0, 0,
     0, 7, 0, 0, 6, 0, 0, 0, 0])
```

图 6.43 所示是输出结果。

```
[4, 1, 9, 6, 2, 7, 8, 5, 3]
[8, 3, 6, 1, 5, 4, 7, 2, 9]
[7, 2, 5, 3, 9, 8, 6, 4, 1]
[5, 8, 4, 9, 7, 6, 3, 1, 2]
[2, 9, 3, 5, 8, 1, 4, 7, 6]
[1, 6, 7, 2, 4, 3, 5, 9, 8]
[6, 4, 1, 7, 3, 2, 9, 8, 5]
[3, 5, 8, 4, 1, 9, 2, 6, 7]
[9, 7, 2, 8, 6, 5, 1, 3, 4]
```

图6.43　输出结果

第 7 章

贪心算法

贪心算法就是遵循某种既定原则，不断地选取当前条件下最优的选择来构造每一个子步骤的解决方案，直到获得问题最终的求解。即在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，所做的仅是在某种意义上的局部最优解。

利用贪心算法解题，需要解决以下两个问题。

一是问题是否适合用贪心法求解，即所求解问题是否具有贪心选择性质。所谓贪心选择性质，是指应用同一规则 f，将原问题变为一个相似的但规模更小的子问题，后面的每一步都是当前看似最佳的选择。这种选择依赖于已做出的选择，但不依赖于未做出的选择。从全局来看，运用贪心策略解决的问题在程序的运行过程中无回溯过程。

二是问题是否具有局部最优解，从而通过选择一个贪心标准，可以得到问题的最优解。

利用贪心算法解题的思路一般为：

- 建立对问题精确描述的数学模型，包括定义最优解的模型。
- 将问题分成一系列子问题，同时定义子问题的最优解结构。
- 应用贪心算法原则可以确定每个子问题的局部最优解，并根据最优解模型，用子问题的局部最优解堆叠出全局最优解。

在本章中，我们重点关注以下三个问题：

- 硬币找零问题

- 活动安排问题
- 哈夫曼编码

7.1　硬币找零问题

我们来看一个用最少硬币支付指定额度的问题。

7.1.1　问题描述

如图 7.1 所示，设有 6 种不同面值的硬币，各硬币的面值分别为 5 分、1 角、2 角、5 角、1 元和 2 元。现要用这些面值的硬币来购物和找钱。购物时规定了可以使用的各种面值的硬币个数。

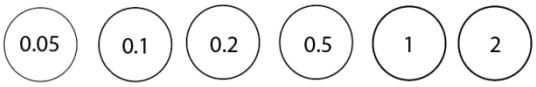


图7.1　硬币

假定商店里各面值的硬币足够多，顾客也可用多种方式支付。在一次购物中希望使用最少硬币个数。举一个例子，一名顾客需要付款 0.55 元，但她身上没有 5 角的硬币。图 7.2 展示了一种情况：顾客用 0.2+0.2+0.1+0.05 共 4 枚硬币来付款。

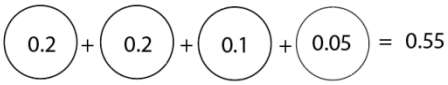


图7.2　方案一

图 7.3 展示了另一种情况，顾客付出 1 元，商店找回 4 角 5 分。这个情况同样需要 4 枚硬币。

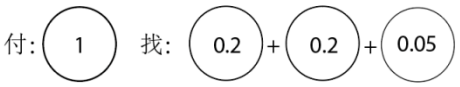


图7.3　方案二

图 7.4 展示了最佳情况。如果顾客付出 1.05 元（1 枚 1 元和 1 枚 5 分），商店找回 5 角，整个过程只需要 3 枚硬币。这个方案用的硬币个数最少。

图7.4　最佳方案

现在问题来了，对于给定的各种面值的硬币个数和付款金额，如何计算使用硬币个数最少的交易方案。

其核心思想为消费者硬币数量有限，商店的硬币无限。因此问题可以用以下公式来描述：

- MIN（消费者支付硬币数量+商店找零硬币数量）
- 支付值-找零值=商品值

对问题的求解转换为寻找上面两个问题的最优解。

这里所用的贪心算法为：MAX（消费者拥有的硬币面值-商店拥有的硬币面值）优先使用。

例如消费者拥有面值为 2 元的硬币，商店拥有 5 分的硬币，因此 MAX=2 元-5 分=195 分。那么上面所有的组合情况为：

2 元（不找零的情况），2 元-5 分，2 元-1 角，……，5 分（不找零的情况）

现在假如购买的商品是 2 元，那么在上面的组合中使用 2 元优先，只用 1 个硬币即可；假如商品是 2 元 9 角 5 分，从上面的序列中使用贪心算法来选择，那么就是一枚 2 元的硬币优先，那么 9 毛 5 如何处理呢，因为上面的序列中存在“1 元-5 分”的情况，这是能够使用的最大币值（贪心），就用它了，那么它是 2 枚硬币，即支付 1 元找零 5 分，总共用了 3 枚硬币。

接下来验证这个算法的贪心选择性和最优子结构性质，证明贪心算法可以获得最优解。

贪心选择性：用贪心策略的选择替代原来的选择，如果能获得更优解或同优解，那么贪心策略就可以获得最优解。

我们把相应的序列列出来（用“分”来表示）：

C[]={200,195,190,180,150,100,95,90,80,50,45,40,30,20,15,10,5}；

证明这个序列具有贪心选择性质即可，这里我们先用一个简单的例子来证明。

c[]={10,5,2,1}，这里，c[i]的出现次数最多为｛n，1,2,1｝，这样才能得到最优解，因为面值为 1 的次数为 2 时，可以用一个面值为 2 的来代替，硬币使用数就少了，那么就可以获得更优解。面值为 2 的硬币只能选用 2 个，若选用了 3 个，可以用 1 枚面值为 5 和一枚面值为 1 的硬币来代替。概括来讲，如果 c[i]超过一定数量，可以用更少的 c[i-1]来代替，因此，优先选择大的面值的话，就会用更少的硬币数，因此满足贪心选择性质。

这里回归到问题的序列 C[]，容易看出，C[i]总可以用更大的 C[i-1]或者配合一些小币值来代替，但是数量<=C[i]的使用数量，因此满足贪心选择性质。当然在这个过程中要考虑消费者硬币是否足够的问题。

最优子结构的性质证明：设 s[i]为各硬币使用数量，S[i]是商品价格为 n 时的最优解，硬币数为 K。现在设某面值的硬币数量减 1：S[j]=S[j]−1；则新的 S[i]是 n−c[j]的最优解，硬币数为 k−1，否则，设 T[i]是 n−c[j]的最优解，硬币数为 m，即 m<k−1；那么对于 n 来说，T[i]的硬币数加 1 应该是最少的硬币数，即 m+1<k−1+1=k，与 k 是最少硬币数矛盾，故问题满足最优子结构性质。

7.1.2　最终代码

找零问题的代码如代码 7.1 所示。

代码 7.1　找零问题的贪心算法求解

```
d = [0.05,0.1,0.2,0.5,1,2]          # 存储每种硬币面值
d_num = []                          # 存储每种硬币的数量
s = 0
# 拥有的零钱总和
temp = raw_input('请输入每种零钱的数量: ')
d num0 = temp.split(" ")
for i in range(0, len(d_num0)):
    d_num.append(int(d_num0[i]))
    s += d[i] * d_num[i]            # 计算出收银员拥有多少钱
sum = float(raw_input("请输入需要找的零钱:"))
if sum > s:
    # 当输入的总金额比收银员的总金额多时，无法进行找零
    print("数据有错")
    return 0
s = s - sum
# 要想用的硬币数量最少，那么需要利用面值大的硬币，
# 因此从数组的面值大的元素开始遍历
i = 6
while i >= 0:
    if sum >= d[i]:
        n = int(sum / d[i])
        if n >= d_num[i]:
            n = d_num[i]            # 更新 n
        sum -= n * d[i]             # 贪心的关键步骤，令 sum 动态改变
        print("用了%d 个%f 元硬币"%(n, d[i]))
    i -= 1
```

7.2　活动安排问题

活动安排问题是解决需要共享公共资源的一系列活动的高效安排问题，以在限定的资源前提下尽可能多地安排活动。

7.2.1 问题描述

有若干个活动，第 i 个活动的开始时间和结束时间是[Si, fi)，只有一间教室，活动之间不能交叠，求最多能安排多少个活动？

分析：目的是提高教室的利用率，尽可能多地安排活动。

考虑容易想到的几种贪心策略。

开始最早的活动优先，目的是想尽早结束活动，让出教室。然而，这个显然不行，因为最早的活动可能时间很长，影响后面的活动。例如，活动开始和结束时间分别为[0, 100)、[1,2)、[2, 3)、[3, 4)、[4, 5]，安排［0, 100)这个活动之后，无法安排其他活动，最优解是安排除它之外的 4 个活动。

短活动优先，目的也是尽量空出教室。但是不难构造如下反例： [0,5)、[5,10)、[3, 7)，这里[3,7)最短，但如果安排了[3,7)，其他两个无法安排了。最优解显然是安排其他两个，而放弃[3,7)，可见这个贪心策略也是不行的。

最少冲突的活动优先，既然上面说安排活动是想减少冲突，那么如果优先安排冲突最少的活动呢？至少从前两个例子来看，这个策略是有效的。真是对的吗？尝试一下下面这个例子：

```
[0,2) [2,4) [4,6) [6,8)
[1,3) [1,3) [1,3) [3,5) [5,7) [5,7) [5,7)
```

[0,2)和 3 个活动冲突——3 个[1,3)，[2,4)和 4 个活动冲突——3 个[1,3)和一个[3,5)，[4,6)和也和 4 个活动冲突——3 个[5,7)和一个[3,5)，[6,8)和 3 个活动冲突——3 个[5,7)。

下面的[1,3)和[5,7)每个都和 5 个活动冲突，而[3,5)只和两个活动冲突——[2,4)和[4,6)。

按照我们的策略应该先安排[3,5)，可是一旦选择了[3,5)，最多只可能安排 3 个活动。

但明显第一行的 4 个活动都可以安排下来，所以这种策略也是不对的。

看似最不对的策略——结束时间越早的活动优先，反而是有效的。可以这样证明，假设最优解 OPT 中安排了 m 个活动，我们把这些活动按照结束时间由小到大排序，显然是不冲突的。假设排好顺序后，这些活动是 $a(1)$、$a(2)$、$a(3)$、…、$a(m)$。

按照我们的贪心策略，选出的活动自然是按照结束时间排好顺序的，并且也都是不冲突的，这些活动是 $b(1)$、$b(2)$、…、$b(n)$。

关键问题是，假设 $a(1) = b(1)$、$a(2) = b(2)$、…、$a(k) = b(k)$，但是 $a(k+1) != b(k+1)$，回答以下几个问题：

- $b(k+1)$会在 $a(k+2)$、$a(k+3)$、…、$a(m)$中出现吗？

  不会。因为 $b(k+1)$的结束时间是最早的，即 $f(b(k+1)) <= f(a(k+1))$，而 $a(k+2)$、$a(k+3)$、…、$a(m)$的开始时间和结束时间都在 $f(a(k+1))$之后，所以 $b(k+1)$不在其中。

- $b(k+1)$和 $a(1)$、$a(2)$、…、$a(k)$冲突吗？

  不冲突，因为 $a(1)$、$a(2)$、…、$a(k)$就是 $b(1)$、$b(2)$、…、$b(k)$。

- $b(k+1)$和 $a(k+2)$、$a(k+3)$、…、$a(m)$冲突吗？

  不冲突，因为 $f(b(k+1)) <= f(a(k+1))$，而 $a(k+2)$、$a(k+3)$、…、$a(m)$的开始时间都在 $f(a(k+1))$之后，更在 $f(b(k+1))$之后。

因此可以把 $a(k+1)$ 换成 $b(k+1)$，从而最优解和贪心得到的解多了一个相同的，经过一个一个替换，可以把最优解完全替换成贪心策略得到的解，从而证明了这个贪心策略的最优性。

7.2.2　最终代码

通过比较下一个活动的开始时间与上一个活动的结束时间的大小关系，确定这两个活动是否是相容的，如果开始时间大于结束时间，则相容，反之不相容，程序如代码 7.2 所示。

代码 7.2　活动安排问题的贪心算法求解

```
#用冒泡排序法对结束时间进行排序，同时得到对应的开始时间的 list
def bubble_sort(s,f):
    for i in range(len(f)):
        for j in range(0,len(f)-i-1):
            if f[j] > f[j+1]:
                f[j], f[j+1] = f[j+1],f[j]
                s[j],s[j+1] = s[j+1],s[j]
    return s,f

def greedy(s,f,n):
    a = [True for x in range(n)]
    #初始选择第一个活动
    j = 0
    for i in range(1,n):
        #如果下一个活动的开始时间大于等于上一个活动的结束时间
        if s[i] >= f[j]:
            a[i] = True
            j = i
        else:
```

```
                a[i] = False
    return a

    n = int(input())
    arr = input().split()
    s = []
    f = []
    for ar in arr:
        ar = ar[1:-1]
        start = int(ar.split(',')[0])
        end = int(ar.split(',')[1])
        s.append(start)
        f.append(end)

    s,f = bubble sort(s,f)
    A = greedy(s,f,n)

    res = []
    for k in range(len(A)):
        if A[k]:
            res.append('({},{})'.format(s[k],f[k]))
    print(' '.join(res))
```

7.3 哈夫曼编码

哈夫曼编码是一种字符编码方式，可以对指定的字符集进行数据压缩，压缩率在 20%~90%。

7.3.1 问题描述

现在有一个包含 5 个字符的字符表{A, B, C, D, E}，各个字符出现的频率统计如表 7.1 所示。

表 7.1 各个字符出现的频率

字符	A	B	C	D	E
出现概率	0.35	0.1	0.2	0.2	0.15

需要构造一种有效率的编码类型，使用该编码表达以上字符表内容时可以产生平均长度最短的位串。

在对由 n 个字符组成的文本进行编码的过程中，有两种编码方式，即定长编码和变长编码。

对于定长编码而言，会为每个字符赋予一个长度固定为 m（$m >= \log_2 n$）的位串，我们常用的标准 ASCII 码就是采用定长编码策略对字符集进行编码的。

长度各异的编码，其中出现频率较高的字符，采用长度较短的编码表示，出现频率较低的字符，采用长度较长的编码表示。著名的摩斯电码（Morse code）就是采用这种策略进行编码的。

通常情况下，与定长编码相比，变长编码可以有效减少表示同一字符集所需的编码长度，提升编码效率。

但是，为了使用变长编码策略，需要解决在定长编码模式下不会遇到的一个问题，就是前缀码问题。所谓前缀码，是对每一个字符规定一个 0-1 串作为其代码，并要求任一字符的代码都不是其他字符代码的前缀，这种编码称为前缀码。

有了前缀码，可以在编码完成的位串中准确定位每个属于字符集的字符。通过简单扫描一个位串，直到得到某个等于字符集中字符的位串后，将该字符替换之前的位串，重复以上操作，即可根据位串恢复原来的文本。

7.3.2　哈夫曼树

为了对某字母表构造一套二进制的前缀码，可以借助二叉树。将树中所有的左向边都标记为 0，所有的右向边都标记为 1。通过记录从根节点到字符所在的叶子节点的简单路径上的所有 0-1 标记来获得表示该字符的编码。

根据二叉树的性质，因为从一个叶子节点到另一个叶子节点的简单路径不存在，所以字符集中的每个字符，其对应的编码不可能是其他字符的前缀，因此，任何一棵这样的二叉树均能生成一套前缀码。

对于给定的字符集和字符表，可以确定每个字符的出现频率，我们怎么才能构造一棵二叉树，将较短的编码分配给高频字符，将较长的编码分配给低频字符呢？

用贪心算法可以实现这个目标。这个算法由戴维·哈夫曼（David Huffman）发明，因此，能达到这个目标的二叉树称为哈夫曼树。

具体算法如下：

- 初始化 n 个单节点的树，并为它们标上字母表中的字符。把每个字符出现的频率记在其对应的根节点中，用来标记各个树的权重，即树的权重等于树中所有叶子节点的概率之和。
- 重复下面的步骤，直到只剩一颗单独的树。找到两棵权重最小的树，若两棵树权重相同，可任选其一，分别把它们作为新二叉树的左右子树，并把其权重之和作为新的权重记录在新树的根节点中。

用上述算法构造出的二叉树即为哈夫曼树。根据哈夫曼树获取的编码称为哈夫曼编码。下面构造 7.3.1 节中提到的字符集的哈夫曼树。

在初始状态，构造如图 7.5 所示的根节点集合。

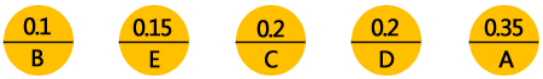


图7.5　初始化根节点的示意图

此时，我们将节点 B 和 E 合并成一棵新的子树，如图 7.6 所示。

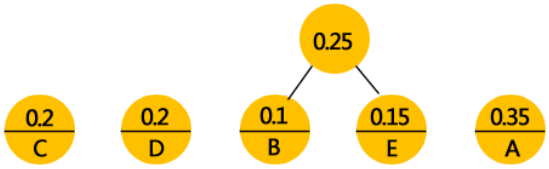


图7.6　合并节点B和E后的示意图

合并节点 B 和 E 之后，重新排序根节点，合并两个权值最小的节点，C 和 D，如图 7.7 所示。

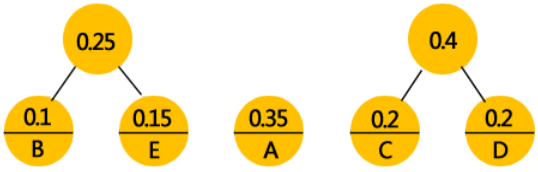


图7.7　合并节点C和D后的示意图

之后，合并由节点 B 和 D 组成的树的根节点和节点 A，如图 7.8 所示。

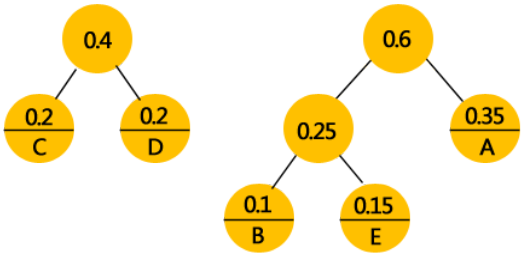


图7.8　合并由节点B和D组成的树的根节点和节点A后的示意图

最后，获取剩余的两个节点，得到结果哈夫曼树，如图 7.9 所示。

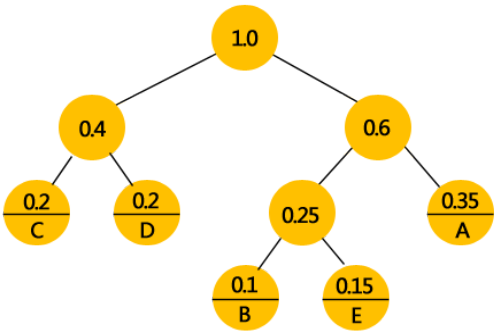


图 7.9　结果哈夫曼树示意图

获得哈夫曼树后，根据给结果哈夫曼树的左侧分支编 0，右侧分支编 1，可以得到字符表中的各个字符的对应编码，如表 7.2 所示。

表 7.2　对应编码

字符	A	B	C	D	E
出现概率	0.35	0.1	0.2	0.2	0.15
哈夫曼编码	11	100	00	01	101

至此，7.3.1 节的字符集编码问题得到解决。根据给定的字符出现的概率和求得的各字符对应的编码长度，若采用上述编码策略，每个字符的平均位长度为：

$$2\times0.35+3\times0.1+2\times0.2+2\times0.2+3\times0.15=2.25$$

若采用定长编码策略，则表示 7.3.1 节的字符表中的每个字符至少需要用 3 位编码来表示。因此，针对这一实例，哈夫曼编码的实现的压缩率为：

$$[(3-2.25)/3]\times100\%=25\%$$

综上，若采用哈夫曼编码，我们表达该字符集时比采用定长编码策略可以少占用 25%的存储空间。

7.3.3　贪心选择性质

二叉树 T 表示字符集 C 的一个最优前缀码，证明可以对 T 做适当修改后得到一棵新的二叉树 T"。在 T"中，x 和 y 是最深叶子节点且为兄弟，同时 T"表示的前缀码也是 C 的最优前缀码。设 b 和 c 是二叉树 T 的最深叶子节点，且为兄弟。设 $f(i)$ 为节点 i 对应字符出现的频率，且 $f(b)<=f(c)$，$f(x)<=f(y)$。

由于 x 和 y 是 C 中具有最小频率的两个字符，有 $f(x)<=f(b)$，$f(y)<=f(c)$。首先，在树 T 中交换叶子 b 和 x 的位置得到 T'，然后在树 T'中交换叶子 c 和 y 的位置，得到树 T''，如图 7.10 所示。

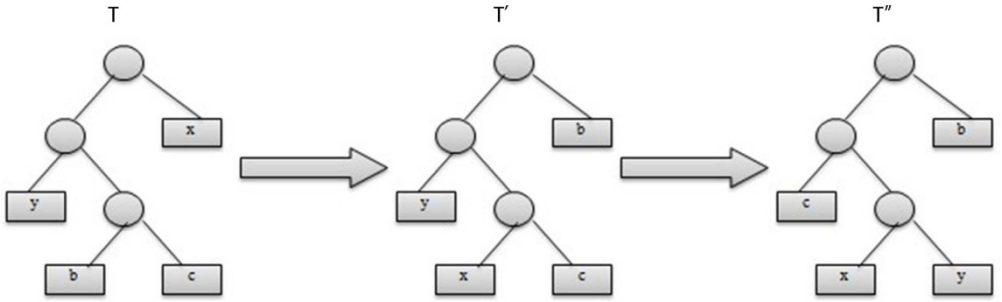


图7.10　贪心选择性质证明示意图

由此可知，树 T 和 T'的前缀码的平均码长之差如下：

$$
\begin{aligned}
&B(T)-B(T')=\sum_{c\in C}f(c)d_T(c)-\sum_{c\in C}f(c)d_{T'}(c)\\
&=f(x)d_T(x)+f(b)d_T(b)-f(x)d_{T'}(x)-f(b)d_{T'}(b)\\
&=f(x)d_T(x)+f(b)d_T(b)-f(x)d_T(b)-f(b)d_T(x)\\
&=(f(b)-f(x))(d_T(b)-d_T(x))\geqslant 0
\end{aligned}
$$

$$
\left.\begin{aligned}
&B(T'')\leqslant B(T')\leqslant B(T)\\
&B(T'')\geqslant B(T)
\end{aligned}\right\}B(T'')=B(T)
$$

因此，T''表示的前缀码也是最优前缀码，且 x 和 y 具有相同的码长，同时，仅最优一位编码不同。

7.3.4　最优子结构性质

二叉树 T 表示字符集 C 的一个最优前缀码，x 和 y 是树 T 中的两个叶子节点且为兄弟，z 是它们的父亲。若将 z 当作具有频率 $f(z)=f(x)+f(y)$的字符，则树 T'=T-{x,y}表示字符集 $C'=C-\{x,y\}\cup\{z\}$的一个最优前缀码。因此，有：

$$
\begin{aligned}
&(1)\ c\in C-\{x,y\},d_T(c)=d_{T'}(c)\Rightarrow f(c)d_T(c)=f(c)d_{T'}(c)\\
&(2)\ f(x)d_T(x)+f(y)d_T(y)=f(x)+f(y)+f(z)d_{T'}(z)
\end{aligned}
$$

$$
B(T)=B(T')+f(x)+f(y)
$$

如果 T'不是 C'的最优前缀码，假定 T''是 C'的最优前缀码，那么显然 T''是比 T 更优的前缀

码，跟前提矛盾！故 T'所表示的 *C*'的前缀码是最优的。

由贪心选择性质和最优子结构性质可以推出哈夫曼算法是正确的，即 HuffmanTree 产生的一棵最优前缀编码树。

7.3.5　最终代码

哈夫曼编码的贪心算法代码如代码 7.3 所示。

代码 7.3　哈夫曼编码的贪心算法求解

```
#Huffman Encoding

#Tree-Node Type
class Node:
    def   init  (self,freq):
        self.left = None
        self.right = None
        self.father = None
        self.freq = freq
    def isLeft(self):
        return self.father.left == self
#创建叶子节点
def createNodes(freqs):
    return [Node(freq) for freq in freqs]

#创建 Huffman 树
def createHuffmanTree(nodes):
    queue = nodes[:]
    while len(queue) > 1:
        queue.sort(key=lambda item:item.freq)
        node left = queue.pop(0)
        node right = queue.pop(0)
        node father = Node(node left.freq + node right.freq)
        node father.left = node left
        node father.right = node right
        node left.father = node father
        node right.father = node father
        queue.append(node father)
    queue[0].father = None
    return queue[0]
#Huffman 编码
def huffmanEncoding(nodes,root):
```

```
    codes = [''] * len(nodes)
    for i in range(len(nodes)):
        node_tmp = nodes[i]
        while node_tmp != root:
            if node_tmp.isLeft():
                codes[i] = '0' + codes[i]
            else:
                codes[i] = '1' + codes[i]
            node_tmp = node_tmp.father
    return codes

if __name__ == '__main__':
    #chars = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N']
    #freqs = [10,4,2,5,3,4,2,6,4,4,3,7,9,6]
    chars_freqs = [('C', 2), ('G', 2), ('E', 3), ('K', 3), ('B', 4),
                   ('F', 4), ('I', 4), ('J', 4), ('D', 5), ('H', 6),
                   ('N', 6), ('L', 7), ('M', 9), ('A', 10)]
    nodes = createNodes([item[1] for item in chars_freqs])
    root = createHuffmanTree(nodes)
    codes = huffmanEncoding(nodes,root)
    for item in zip(chars_freqs,codes):
        print 'Character:%s freq:%-2d  encoding: %s' % (item[0][0],item[0][1],item[1])
```

最终的输出结果为：

```
>>>
Character:C freq:2  encoding: 10100
Character:G freq:2  encoding: 10101
Character:E freq:3  encoding: 0000
Character:K freq:3  encoding: 0001
Character:B freq:4  encoding: 0100
Character:F freq:4  encoding: 0101
Character:I freq:4  encoding: 0110
Character:J freq:4  encoding: 0111
Character:D freq:5  encoding: 1011
Character:H freq:6  encoding: 1110
Character:N freq:6  encoding: 1111
Character:L freq:7  encoding: 001
Character:M freq:9  encoding: 100
Character:A freq:10 encoding: 110
```

第 8 章

动态规划算法

动态规划算法将待求解问题拆分成一系列相互交叠的子问题，通过递推关系定义各子问题的求解策略，并随时记录子问题的解，最终获得原始问题的解，避免了对交叠子问题的重复求解。

在动态规划算法中有三要素，即最优子结构、边界和状态转移函数。最优子结构是指每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到；边界是指问题最小子集的解；状态转移函数是指从一个阶段向另一个阶段过渡的具体模式，描述的是两个相邻子问题之间的关系。

具备以上三要素的问题均可以采用动态规划的策略进行求解，在本章中，我们重点关注以下 4 个问题：

- 爬楼梯问题
- 矿工挖矿问题
- 背包问题
- 最长递归子序列问题

8.1 爬楼梯问题

我们首先用动态规划方法解决日常生活中经常遇到的一个问题——爬楼梯问题。

8.1.1 问题描述

小明家住在二楼，每次回家都需要经过一个有 10 层台阶的楼梯。小明每次可以选择一步走一级台阶或者一步走两级台阶。请帮小明计算他从楼下到家一共有多少种走法。

打个比方，小明可以选择每次都走一级台阶，那么他回家一共需要走十步，这是其中的一种走法。不然，他还可以选择一次走两级台阶，那么他回家一共需要走五步，这是另外一种走法。除此之外，还有很多种不同的走法，现在我们要做的是把所有可能的走法数量统计出来。

在正式写程序之前，我们先一起整理一下该问题的解题思路。

首先考虑小明走最后一步的情况，他要么是从第九级台阶再走一级到第十级，要么是从第八级台阶走两级到第十级，如图 8.1 所示。也就是说，要想到达第十级台阶，最后一步一定是从第八级或者第九级台阶开始的。

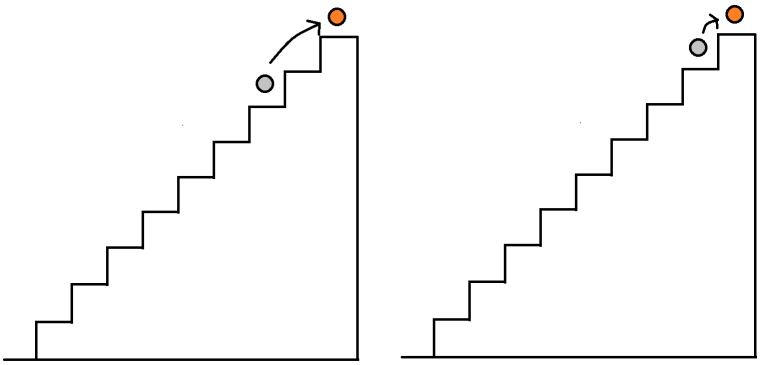


图8.1　到达第十级台阶的前一步可选的方案

也就是说，如果已知从地面到第八级台阶一共有 X 种走法，从地面到第九级台阶一共有 Y 种走法，那么从地面走到第十级台阶一定是 X 和 Y 之和。

为了描述方便，用 $F(n)$表示第 n 级台阶的走法数量，根据以上分析，可以得到 $F(10)=F(9)+F(8)$。推而广之，对于 $F(n)$，我们可以通过 $F(n-1)$和 $F(n-2)$计算得到，即 $F(n)=F(n-1)+F(n-2)$。

当问题继续细化，即只有一级和两级台阶时，我们可以直接得出结论，无须继续细化，即 $F(1)=1$，$F(2)=2$。如图 8.2 所示，为到达第二级台阶的走法示例。

分析到这里，该问题作为动态规划问题求解的三个要素就全部出现了，如下所述。

- 边界：$F(1)=1$，$F(2)=2$。
- 最优子结构：$F(n)$的最优子结构即 $F(n-1)$和 $F(n-2)$。
- 状态转移函数：$F(n)=F(n-1)+F(n-2)$。

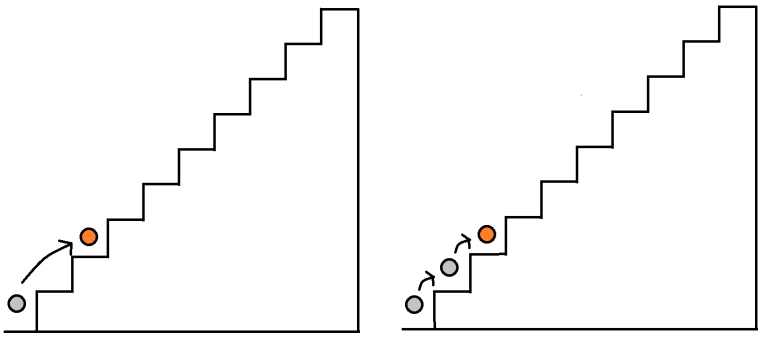


图8.2　到达第二层台阶的可选方案

至此，爬楼梯问题的动态规划建模过程已经完成。

考虑到算法的时间和空间的效率问题，为了避免重复计算子问题的解，我们利用最优子结构特性，采用自底向上的方式进行计算，具体过程如下所述。

根据边界定义，我们已经知道前两级台阶的走法数量，如表 8.1 所示。

表 8.1　前两级台阶的走法数量对照表

台阶数量	1	2	3	4	5	6	7	8	9	10
走法数量	1	2								

表 8.1 中第一行表示楼梯的台阶数量，第二行中对应的各列表示针对各级台阶可能的走法数量。在后续的求解过程中，我们可以利用表格中已知的子问题的解，根据状态转移函数，获得新的阶段的子问题的解，通过迭代，不断将表格填满，从而得到该问题的最终解。

在第一次迭代过程中，台阶数量为 3，根据状态转移函数，可知 $F(3)=F(2)+F(1)=3$，因此目前求解状态如表 8.2 所示。

表 8.2　第三级台阶的走法数量对照表

台阶数量	1	2	3	4	5	6	7	8	9	10
走法数量	1	2	3							

在第二次迭代过程中，台阶数量为 4，根据状态转移函数，可知 $F(4)=F(3)+F(2)=5$，因此目前求解状态如表 8.3 所示。

表 8.3　第四级台阶的走法数量对照表

台阶数量	1	2	3	4	5	6	7	8	9	10
走法数量	1	2	3	5						

依次类推，经过八次迭代之后，即可获得第十级台阶的走法数量，如表 8.4 所示。

表 8.4　各级台阶的走法数量对照表

台阶数量	1	2	3	4	5	6	7	8	9	10
走法数量	1	2	3	5	8	13	21	34	55	89

总结上面的过程，对每一级楼梯的走法数量而言，其求解过程所依赖的都是其前一级和前两级楼梯的走法数量。因此，在每一次迭代过程中，仅需要关注这两个变量的取值，即可获得所求的新的变量的值。

现在来把这个过程转化为程序。

定义函数 upstairs(n)来计算第 n 级台阶的走法数量。在正式迭代之前，首先界定边界的情况。当 n 小于 1 时，说明该输入变量不合法，返回的走法数量为 0；当 n 为 1 时，即为第一种边界的情况，返回值为 1；当 n 为 2 时，即为第二种边界的情况，返回值为 2。

```
01  def upstairs(n):
02      if  n < 1:                  #判断当前台阶级数是否小于 1
03          print(0)
04      if  n == 1:                 #判断当前台阶级数是否为 1
05          print(1)
06      if  n == 2:                 #判断当前台阶级数是否为 2
07          print(2)
08
09      a = 1
10      b = 2
11      temp = 0
12      for i in range(3,n):        #迭代求解各级台阶的走法数量
13          temp = a + b
14          a = b
15          b = temp
16      print(temp)
```

之后即进入循环的迭代过程，在迭代中，根据之前的分析，仅需要关注 $n-1$ 和 $n-2$ 的取值，即可通过状态转移函数 $F(n)=F(n\text{-}1)+F(n\text{-}2)$获得 $F(n)$的值。因此，整个迭代过程仅需要引入一个临时变量循环保存当前子问题的解即可。

由此可见，该解决方案的时间复杂度为 $O(n)$，而空间复杂度只有 $O(1)$。

8.1.2　最终代码

爬楼梯问题的动态规划求解的程序如代码 8.1 所示。

代码 8.1　爬楼梯问题的动态规划求解

```
a = 1                          #初始化边界值
b = 2
temp = 0

def upstairs(n):
    if  n < 1:                 #判断当前台阶级数是否小于 1
        print(0)
    if  n == 1:                #判断当前台阶级数是否为 1
        print(1)
    if  n == 2:                #判断当前台阶级数是否为 2
        print(2)

    for i in range(3,n):       #迭代求解各级台阶的走法数量
        temp = a + b
        a = b
        b = temp
    print(temp)                #输出计算结果
```

首先定义边界值，确定 $F(1)$和 $F(2)$的取值。之后，通过函数 upstairs(n)迭代计算到达各层的楼梯走法数量并输出，最终获得问题的解。

8.2　矿工挖矿问题

矿工挖矿问题是为了解决在给定金矿和矿工数量的前提下，能够获得最多黄金的挖矿策略。

8.2.1　问题描述

某一地区发现了 5 座金矿，每座金矿的黄金储量不同，需要参与挖掘的工人数也不同。假设参与挖矿工人的总数是 10 人，且每座金矿要么全挖，要么不挖，不能派出一半人挖取一半金矿。要求用程序求解，要想得到尽可能多的黄金，应该选择挖取哪几座金矿？

金矿的黄金储量与所需挖掘工人的数量如表 8.5 所示。

表 8.5　黄金储量与所需工人数量表

金矿编号	黄金储量	所需工人数量
1	400	5
2	500	5
3	200	3
4	300	4

续表

金矿编号	黄金储量	所需工人数量
5	350	3

如果要用动态规划算法解决该问题，需要定位用于解题的三要素：最优子结构、边界和状态转移函数。

首先寻找最优子结构。我们的解题目标是确定 10 个工人挖 5 座金矿时能够获得的最多的黄金数量，该结果可以从 10 个工人挖 4 座金矿的子问题中递归求解。

在解决 10 个工人挖 4 座金矿的过程中，存在两种选择，一种是放弃第 5 座金矿，将 10 个工人全部投放到前 4 座金矿的挖掘中，如图 8.3 所示。

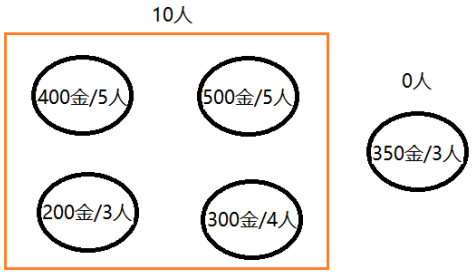


图8.3　放弃第5座金矿的情况

另一种选择是对第 5 座金矿进行挖掘，因此需要从 10 人中分配 3 个人加入第 5 座金矿的挖掘工作中，如图 8.4 所示。

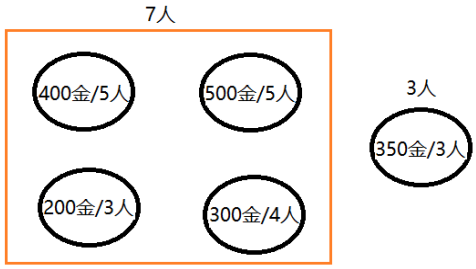


图8.4　挖掘第5座金矿的情况

因此，最终的最优解应该是这两种选择中获得黄金数量较多的那个，即为图 8.3 所描述的场景与图 8.4 所描述场景中的最大值。

为了方便描述，假设金矿的数量为 n，工人的数量为 w，当前获得的黄金数量为 $G[n]$，当前所用矿工的数量为 $P[n]$，则根据上述分析，要获得 10 个矿工挖掘第 5 座金矿的最优解 $F(5, 10)$，

需要在 $F(4, 10)$和 $F(4, 10-P[4])+G[4]$中获取较大的值，即

$$F(5, 10) = \max(F(4, 10), F(4, 10-P[4])+G[4])$$

因此，针对该问题而言，以上便是 $F(5, 10)$情况下的最优子结构。

之后，我们来考虑该问题的边界。对于一座金矿的情况，若当前的矿工数量不能满足该金矿的挖掘需要，则获得的黄金数量为 0，若能满足矿工数量要求，则获得的黄金数量为 $G[0]$。因此，该问题的边界条件可表述为：

当 $n=1, w>=P[0]$时，$F(n, w)=G[0]$；

当 $n=1, w<P[0]$时，$F(n, w)=0$。

综上，可以得到该问题的状态转移函数：

$$F(n,w) = 0 \ (n<=1, w<p[0])$$
$$F(n,w) = G[0] \ (n==1, w>=P[0])$$
$$F(n,w) = F(n-1,w) \ (n>1, w<P[n-1])$$
$$F(n,w) = \max(F(n-1,w), F(n-1,w-P[n-1])+G[n-1]) \ (n>1, w>=P[n-1])$$

至此，定义了用动态规划算法解决该问题的三个要素，下面要做的是利用边界、最优子结构和状态转移函数对该问题进行求解。

在初始化阶段，利用表格分析求解思路。如表 8.6 所示，表格的第一列代表挖掘金矿数，即 n 的取值情况；表格的第一行代表占用工人数，即 w 的取值情况；中间各空白区域是需要通过计算填入的对应的黄金数量，即 $F(n, w)$的取值。

表 8.6　初始黄金数量表

w / *n*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿										
2 金矿										
3 金矿										
4 金矿										
5 金矿										

在挖掘第一座金矿时，由于其所需的工人数量为 5，所以当 w 的取值小于 5 时，根据公式 $F(n,w) = 0 \ (n<=1, w<p[0])$，获得的黄金数量均为 0。当 w 的取值大于或等于 5 时，根据公式 $F(n,w) = G[0] \ (n==1, w>=p[0])$，黄金数量的取值为 400，如表 8.7 所示。此时确定了该问题的边界。

表 8.7　挖掘第 1 座金矿时的黄金数量表

w / *n*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿	0	0	0	0	400	400	400	400	400	400
2 金矿										
3 金矿										
4 金矿										
5 金矿										

在挖掘第 2 座金矿时，由于其需要 5 个人进行挖掘，因此当 w 取值小于 5 时，根据公式 $F(n,w) = F(n-1,w)$ $(n>1, w<P[n-1])$，$F(2, w) = F(1, w) = 0$；当 w 取值大于或等于 5 时，根据公式 $F(n,w) = \max(F(n-1,w), F(n-1,w-P[n-1])+G[n-1])$ $(n>1, w>=P[n-1])$，在 5~9 人的区间里，获得的黄金数量为 500，即所有人都去参加第 2 座金矿的挖掘时获得的黄金量，在参与人数为 10 时，可以两座金矿同时挖掘，因此获得的黄金数量为 900，如表 8.8 所示。

表 8.8　挖掘第 2 座金矿时的黄金数量表

w / *n*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿	0	0	0	0	400	400	400	400	400	400
2 金矿	0	0	0	0	500	500	500	500	500	900
3 金矿										
4 金矿										
5 金矿										

同理，在挖掘第 3 座金矿时，黄金产出量为 200，需要的工人数量为 3，根据上述计算方式可得黄金产出量，如表 8.9 所示。

表 8.9　挖掘第 3 座金矿时的黄金数量表

w / *n*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿	0	0	0	0	400	400	400	400	400	400
2 金矿	0	0	0	0	500	500	500	500	500	900
3 金矿	0	0	200	200	500	500	500	700	700	900
4 金矿										
5 金矿										

第 4 座金矿的黄金产出量为 300，需要的工人数量为 4，根据上述计算方式，可得黄金产出

量，如表 8.10 所示。

表 8.10　挖掘第 4 座金矿时的黄金数量表

n \ *w*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿	0	0	0	0	400	400	400	400	400	400
2 金矿	0	0	0	0	500	500	500	500	500	900
3 金矿	0	0	200	200	500	500	500	700	700	900
4 金矿	0	0	200	300	500	500	500	700	800	900
5 金矿										

针对第 5 座金矿的黄金产出量计算与上述过程一致，具体产出量如表 8.11 所示。

表 8.11　挖掘第 5 座金矿时的黄金数量表

n \ *w*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿	0	0	0	0	400	400	400	400	400	400
2 金矿	0	0	0	0	500	500	500	500	500	900
3 金矿	0	0	200	200	500	500	500	700	700	900
4 金矿	0	0	200	300	500	500	500	700	800	900
5 金矿	0	0	350	350	500	550	650	850	850	900

通过以上的计算过程，我们不难发现，除了第 1 座金矿的相关数据，表格中的其他数据都可以由前一行的一个或两个格子中的数推导而来。例如，3 座金矿 8 个人挖掘的黄金量 $F(3, 8)$ 就来自 2 座金矿 8 个人挖掘的黄金量 $F(2, 8)$和 2 座金矿 5 个人挖掘的黄金量 $F(2, 5)$，即 $F(3, 8) = \max(F(2, 8), F(2, 5) + 200) = \max(500, 500 + 200) = 700$。

再比如，4 座金矿 10 个人挖掘的黄金量 $F(4，10)$，来自 3 座金矿 10 个人挖掘的黄金量 $F(3，10)$和 3 座金矿 6 个人挖掘的黄金量 $F(3，6)$，即 $F(4，10) = \max(F(3，10)，F(3，6) + 300) = \max(900，500 + 300) = 900$，如表 8.12 所示。

表 8.12　2–5 座金矿黄金产量计算依赖于之前的计算量

n \ *w*	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
1 金矿	0	0	0	0	400	400	400	400	400	400
2 金矿	0	0	0	0	500	500	500	500	500	900

续表

n \ w	1 人	2 人	3 人	4 人	5 人	6 人	7 人	8 人	9 人	10 人
3 金矿	0	0	200	200	500	500	500	700	700	900
4 金矿	0	0	200	300	500	500	500	700	800	900
5 金矿	0	0	350	350	500	550	650	850	850	900

根据以上思路，在用程序实现该算法的过程中，采用自底向上的方式进行计算，像填表过程一样从左至右、从上到下逐渐获得计算结果。这样，可以不需要存储整个表格的内容，仅需要存储前一行的结果，就可以推导出下一行的内容。

我们现在来把这个过程转换为程序。

我们定义函数 goldMining(n, w, g, p)来计算挖掘第 n 座金矿、有 w 个工人参与时能获得的黄金量，其中 g 和 p 分别为数组，分别存放对应各座金矿的黄金量及所需工人数。在正式迭代之前，首先界定边界的情况。当工人数量小于 p[0]时，说明目前工人数量不能开始任何金矿的挖掘，获得的黄金数量为 0；当工人数量大于或等于 p[0]时，获得的黄金数量即为 g[0]。

```
01  def goldMining(n, w, g, p):
02      results = []                    #保存返回结果的数组
03      preResults = []                 #保存上一行结果的数组
04      for i in range(0,w):            #填充边界格子的值，从左向右填充表格第一行的内容
05          if  i < p[0]:
06              preResults[i] = 0       #若当前人数少于挖掘第一座金矿所需的人数，黄金量为 0
07          else:
08              preResults[i] = g[0]    #若当前人数不少于第一座金矿所需的人数，黄金量为 g[0]
09
10      for i in range(0,n):            #外层循环为金矿数量
11          for j in range(0,w):        #内层循环为工人数量
12              if j < p[i]:
13                  results[j] = preResults[j]
14              else:
15                  results[j] = Math.max(preResults[j], preResults[j-p[i]] + g[i])
16          preResults = results
17      return results
```

之后即进入循环的迭代过程。在迭代中，根据之前的分析，我们仅需要关注前一行 preResults 的取值，即可通过状态转移函数 $F(n,w) = F(n-1,w)$ $(n>1,\ w<P[n-1])$或 $F(n,w) = \max(F(n-1,w), F(n-1,w-P[n-1])+G[n-1])$ $(n>1,\ w>=P[n-1])$获得 $F(n,w)$的值，因此，整个迭代过程仅需要引入 preResults 数组保存前一行的值即可。

由此可见，该解决方案的时间复杂度为 $O(n*w)$，而空间复杂度只有 $O(w)$。

8.2.2　最终代码

挖金矿问题的动态规划求解代码如代码 8.2 所示。

代码 8.2　挖金矿问题的动态规划求解

```
01  def goldMining(n, w, g, p):
02      results = []                    #保存返回结果的数组
03      preResults = []                 #保存上一行结果的数组
04      for i in range(0,w):            #填充边界格子的值，从左向右填充表格第一行的内容
05          results.append(0)           #初始化结果数组
06          if  i < p[0]:
07              preResults.append(0)  #若人数少于第一座金矿所需人数，黄金量为 0
08          else:
09              preResults.append(g[0])    #若人数不少于第一座金矿所需人数，黄金量为 g[0]
10
11      for i in range(0,n):            #外层循环为金矿数量
12          for j in range(0,w):        #内层循环为矿工数量
13              if j < p[i]:
14                  results[j] = preResults[j]
15              else:
16                  results[j] = Math.max(preResults[j], preResults[j-p[i]] + g[i])
17          preResults = results
18      return results
```

首先定义边界值，确定数组 preResults 中各个元素的取值，同时初始化数组 results。之后，通过函数 goldMining(n, w, g, p)迭代计算各座金矿数量与工人数量组合的黄金量，最终获得问题的解。

8.3　背包问题

本节来看一个景点的动态规划问题——背包问题。

8.3.1　问题描述

有 N 个重量为 $w_1, w_2, ..., w_n$、价值为 $v_1, v_2, ..., v_n$ 的物品和一个承重量为 W 的背包，求让背包里装入的物品具有最大的价值总和的物品子集。

这就是经典的背包问题，可以用动态规划算法进行求解。

总体思路：为了设计一个动态规划算法，需要推导出一个递推关系，用较小子实例的解的形式来表示背包问题的实例的解。

首先考虑一个有前 i（$1 \leqslant i \leqslant N$）个物品定义的实例，物品的重量分别为 $w_1, w_2, ..., w_i$，价值为 $v_1, v_2, ..., v_i$，背包目前的承重量为 j（$1 \leqslant j \leqslant W$）。设 $F(i, j)$为组成该实例最优解的物品的总价值，也就是说，是能够放进承重量为 j 的背包中的前 i 个物品中最有价值的子集的总价值。

在这里，可以把前 i 个物品中能够放进承重量为 j 的背包中的子集分为两种类别：包括第 i 个物品的子集和不包括第 i 个物品的子集。于是我们可以得到以下结论：

- 根据定义，在不包括第 i 个物品的子集中，最优子集的价值是 $F(i-1, j)$。
- 在包括第 i 个物品的子集中（此时，$j-w_i \geqslant 0$），最优子集是由该物品和前 $i-1$ 个物品中能够放进承重量为 $j-w_i$ 的背包的最优子集组成。这种最优子集的总价值等于 $v_i + F(i-1, j-w_i)$。

因此，在前 i 个物品中，最优解的总价值等于以上两种情况中求得的价值的较大值，以上便是该问题的最优子结构。

当然，如果第 i 个物品不能放进背包，从前 i 个物品中选出的最优子集的总价值即等于从前 $i-1$ 个物品中选出的最优子集的总价值。于是，可以得到以下递推式，即状态转移函数：

$$F(i,j)=\begin{cases}\max\{F(i-1,j), v_i+F(i-1,j-w_i)\}, & j-w_i \geqslant 0\\ F(i-1,j), & j-w_i<0\end{cases}$$

同时，可以获得背包问题的边界为：

$$F(0,j)=0, \qquad j \geqslant 0$$
$$F(i,0)=0, \qquad i \geqslant 0$$

至此，该问题的边界、最优子结构和状态转移函数均已求出。

我们的目标是求 $F(N, W)$，即给定 N 个物品中能够放进承重量为 W 的背包的最大总价值以及得到最大总价值的物品组合。

8.3.2 问题实例

根据上述思路，我们来思考以下实例：

假设物品的数量为 4，背包的承重量为 8，各个物品的重量和价值如表 8.13 所示。

表 8.13　重量价值表

i	1	2	3	4
w	2	3	4	5
v	3	4	5	6

1. 原理

动态规划与分治法类似，都是把大问题拆分成小问题，通过寻找大问题与小问题的递推关系，解决一个个小问题，最终达到解决原问题的目的。

但不同的是，分治法在子问题和子子问题上被重复计算了很多次，而动态规划则具有记忆性，通过填写表把所有已经解决的子问题答案记录下来，在新问题里需要用到的子问题可以直接提取，避免了重复计算，从而节约了时间。所以在问题满足最优性原理之后，用动态规划算法解决问题的核心就在于填表，表填写完毕，最优解也就找到了。

2. 过程

把背包问题抽象化（X_1，X_2，…，X_n，其中 Xi 取 0 或 1，表示第 i 个物品选或不选），Vi 表示第 i 个物品的价值，Wi 表示第 i 个物品的体积（重量）。

建立模型，即求 $\max(V_1X_1+V_2X_2+\ldots+V_nX_n)$。

约束条件，$W_1X_1+W_2X_2+\ldots+W_nX_n<\text{capacity}$。

定义 $V(i,j)$，当前背包容量 j，前 i 个物品最佳组合对应的价值。

最优性原理是动态规划的基础。最优性原理是指不依赖于初始状态和初始决策，对于属于最优性决策中的任何一环，其中的任何一个决策都是最优决策。判断该问题是否满足最优性原理，采用反证法证明：

假设$(X_1,X_2,\ldots,X_n)$是背包问题的最优解，则有$(X_2,X_3,\ldots,X_n)$是其子问题的最优解，假设$(Y_2,Y_3,\ldots,Y_n)$是上述问题的子问题的最优解，则理应有$(V_2Y_2+V_3Y_3+\ldots+V_nY_n)+V_1X_1>(V_2X_2+V_3X_3+\ldots+V_nX_n)+V_1X_1$。

而$(V_2X_2+V_3X_3+\ldots+V_nX_n)+V_1X_1=(V_1X_1+V_2X_2+\ldots+V_nX_n)$，则有$(V_2Y_2+V_3Y_3+\ldots+V_nY_n)+V_1X_1>(V_1X_1+V_2X_2+\ldots+V_nX_n)$。

该式子说明$(X_1,Y_2,Y_3,\ldots,Y_n)$才是该背包问题的最优解，这与最开始的假设$(X_1$，X_2，…，$X_n)$是背包问题的最优解相矛盾，故背包问题满足最优性原理。

寻找递推关系式，面对当前商品有两种可能性：

- 包的容量比该商品体积小，装不下，此时的价值与前 i-1 个的价值是一样的，即

$V(i,j)=V(i-1,j)$。

- 还有足够的容量可以装该商品，但装了也不一定达到当前最优价值，所以在装与不装之间选择最优的一个，即 $V(i,j)=\max\{V(i-1,j),V(i-1,j-w(i))+v(i)\}$。

其中 $V(i-1,j)$表示不装，$V(i-1,j-w(i))+v(i)$ 表示装了第 i 个商品，背包容量减少 $w(i)$，但价值增加了 $v(i)$。

由此可以得出递推关系式：

- $j<w(i)$ $V(i,j)=V(i-1,j)$
- $j>=w(i)$ $V(i,j)=\max\{V(i-1,j),V(i-1,j-w(i))+v(i)\}$

填表，首先初始化边界条件，如表 8.14 所示，$V(0,j)=V(i,0)=0$。

表 8.14　初始化边界条件

i/j	0	1	2	3	4	5	6	7	8
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**								
2	**0**								
3	**0**								
4	**0**								

然后一行一行地填表，

- 如，$i=1$，$j=1$，$w(1)=2$，$v(1)=3$，有 $j<w(1)$，故 $V(1,1)=V(1-1,1)=0$。
- 又如 $i=1$，$j=2$，$w(1)=2$，$v(1)=3$，有 $j=w(1)$，故 $V(1,2)=\max\{V(1-1,2)，V(1-1,2-w(1))+v(1)\}$ $=\max\{0，0+3\}=3$。
- 如此下去，填到最后一个，$i=4$，$j=8$，$w(4)=5$，$v(4)=6$，有 $j>w(4)$，故 $V(4,8)=\max\{V(4-1,8)，V(4-1,8-w(4))+v(4)\}=\max\{9, 4+6\}=10$。所以填完如表 8.15 所示。

表 8.15　填表

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	**0**	**3**	**3**	**3**	**3**	**3**	**3**	**3**
2	0	**0**	**3**	**4**	**4**	**7**	**7**	**7**	**7**
3	0	**0**	**3**	**4**	**5**	**7**	**8**	**9**	**9**
4	0	**0**	**3**	**4**	**5**	**7**	**8**	**9**	**10**

填完表格，最优解即是 $V(\text{number,capacity})=V(4,8)=10$，但还不知道解由哪些商品组成，故

要根据最优解回溯找出解的组成，根据填表的原理有如下的寻解方式：

- $V(i,j)=V(i-1,j)$时，说明没有选择第 i 个商品，则回到 $V(i-1,j)$。
- $V(i,j)=V(i-1,j-w(i))+v(i)$时，说明装了第 i 个商品，该商品是最优解组成的一部分，随后我们得回到装该商品之前，即回到 $V(i-1,j-w(i))$。
- 一直遍历到 $i=0$ 为止，所有解的组成都会找到。

如上例子，

- 最优解为 $V(4,8)=10$，而 $V(4,8)!=V(3,8)$却有 $V(4,8)=V(3,8-w(4))+v(4)=V(3,3)+6=4+6=10$，所以第 4 件商品被选中，并且回到 $V(3,8-w(4))=V(3,3)$。
- 有 $V(3,3)=V(2,3)=4$，所以第 3 件商品没被选择，回到 $V(2,3)$。
- 而 $V(2,3)!=V(1,3)$却有 $V(2,3)=V(1,3-w(2))+v(2)=V(1,0)+4=0+4=4$，所以第 2 件商品被选中，并且回到 $V(1,3-w(2))=V(1,0)$。
- 有 $V(1,0)=V(0,0)=0$，所以第 1 件商品没被选择。

到此，背包问题已经解决，如图 8.5 所示，利用动态规划算法解决此问题的效率即是填写此张表的效率，所以动态规划的时间效率为 $O(\text{number*capacity})=O(n*c)$，由于用到二维数组存储子问题的解，所以动态规划的空间效率为 $O(n*c)$。

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

图8.5　问题解决状态示意图

为了提升算法的空间效率，可以进行以下优化：

空间优化，每一次 $V(i)(j)$改变的值只与 $V(i-1)(x)$ $\{x:1...j\}$有关，$V(i-1)(x)$是前一次 i 循环保存下来的值。

因此，可以将 V 缩减成一维数组，从而达到优化空间的目的，状态转移方程转换为 $B(j)=\max\{B(j), B(j-w(i))+v(i)\}$。

并且，状态转移方程，每一次推导 $V(i)(j)$都是通过 $V(i-1)(j-w(i))$来推导的，所以一维数组中 j 的扫描顺序应该从大到小（capacity 到 0），否则前一次循环保存下来的值将会被修改，从而造成错误。

同样以上述例子中 i=3 时来说明，有：

- i=3，j=8，w(3)=4，v(3)=5，有 j>w(3)，则 B(8)=max｛B(8)，B(8−w(3))+v(3)｝=max｛B(8)，B(4)+5｝=max｛7，4+5｝=9。
- j−−即 j=7，有 j>w(3)，则 B(7)=max｛B(7)，B(7−w(3))+v(3)｝=max｛B(7)，B(3)+5｝=max｛7，4+5｝=9。
- j−−即 j=6，有 j>w(3)，则 B(6)=max｛B(6)，B(6−w(3))+v(3)｝=max｛B(6)，B(2)+5｝=max｛7，3+5｝=8。
- j−−即 j=5，有 j>w(3)，则 B(5)=max｛B(5)，B(5−w(3))+v(3)｝=max｛B(5)，B(1)+5｝=max｛7，0+5｝=7。
- j−−即 j=4，有 j=w(3)，则 B(4)=max｛B(4)，B(4−w(3))+v(3)｝=max｛B(4)，B(0)+5｝=max｛4，0+5｝=5。
- j−−即 j=3，有 j<w(3)，继续访问数组会出现越界，所以本轮操作停止，B(0)到 B(3)的值保留上轮循环（i=2 时）的值不变，进入下一轮循环 i++。

如果 j 不逆序而采用正序 j=0...capacity，如表 8.16 和表 8.17 所示，当 j=8 时应该有 B(8)=B(8−w(3))+v(3)=B(4)+5，然而此时的 B(4)已经在 j=4 的时候被修改过了，原来的 B(4)=4，现在 B(4)=5，所以计算得出 B(8)=5+5=10，显然这与正确答案不符；所以该一维数组后面的值需要前面的值进行运算再改动，如果正序遍历，则前面的值将有可能被修改掉从而造成后面数据的错误；相反如果逆序遍历，先修改后面的数据再修改前面的数据，此种情况就不会出错了。

表 8.16　i 取值为 2 和 3 时的情况

i/j	0	1	2	3	4	5	6	7	8
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9

表 8.17　j=0...capacity 的取值情况

i/j	0	1	2	3	4	5	6	7	8
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	10

然而不足的是，虽然优换了动态规划的空间，但是该方法不能找到最优解的解组成，因为动态规划寻找解组成必须要在确定了最优解的前提下再往回找解的构成，而优化后的动态规划只用了一维数组，之前的数据已经被覆盖掉，没办法寻找，所以两种方法各有优缺点。

8.3.3　最终代码

背包问题的动态规划求解代码如代码 8.3 所示。

代码 8.3　背包问题的动态规划求解

```
def bag(n,c,w,v):
    res=[[-1 for j in range(c+1)] for i in range(n+1)]
    for j in range(c+1):
        res[0][j]=0
    for i in range(1,n+1):
        for j in range(1,c+1):
            res[i][j]=res[i-1][j]
            if j>=w[i-1] and res[i][j]<res[i-1][j-w[i-1]]+v[i-1]:
                res[i][j]=res[i-1][j-w[i-1]]+v[i-1]
    return res

def show(n,c,w,res):
    print('最大价值为:',res[n][c])
    x=[False for i in range(n)]
    j=c
    for i in range(1,n+1):
        if res[i][j]>res[i-1][j]:
            x[i-1]=True
            j-=w[i-1]
    print('选择的物品为:')
    for i in range(n):
        if x[i]:
            print('第',i,'个,',end='')
    print('')

if __name__=='__main__':
    n=5
    c=10
    w=[2,2,6,5,4]
    v=[6,3,5,4,6]
    res=bag(n,c,w,v)
    show(n,c,w,res)
```

输出的结果为：

```
最大价值为：15
选择的物品为：
第 0 个，第 1 个，第 4 个
```

8.4 最长递归子序列问题

最长递归子序列问题也是经典的动态规划问题之一。

8.4.1 问题描述

最长递增子序列（Longest Increasing Subsequence，LIS）问题即给定一个序列，求解其中最长的递增的子序列的长度。

要求长度为 i 的序列 $A_i\{a_1,a_2,\cdots,a_i\}$的最长递增子序列，需要先求出序列 $A_{i-1}\{a_1,a_2,\cdots,a_{i-1}\}$中以各元素（$a_1,a_2,\cdots,a_{i-1}$）作为最大元素的最长递增序列，然后把所有这些递增序列与 a_i 进行比较。如果某个长度为 m 的序列的末尾元素 a_j（$j<i$）比 a_i 要小，则将元素 a_i 加入这个递增子序列，得到一个新的长度为 m+1 的新序列，否则其长度不变，将处理后的所有 i 个序列的长度进行比较，其中最长的序列就是要求的最长递增子序列。

举例说明，对于序列 A{3, 1, 4, 5, 9, 2, 6, 5, 0}，当处理到第 7 个元素“6”时，以 3, 1, 4, 5, 9, 2 为最末元素的最长递增序列分别为

3

1

3, 4

3, 4, 5

3, 4, 5, 9

1, 2

因此，“6”可以加入到以“5”作为结尾的序列中，形成以“6”为结尾的最长递增序列{3, 4, 5, 6}。

当新加入第 8 个元素“5”时，这些序列变为

3, 5

1, 5

3, 4, 5

3, 4, 5

3, 4, 5, 9

1, 2, 5

可见此时的以第 8 个元素“5”为结尾的最长递增序列仍为{3, 4, 5}。由于最后一个元素取值为 0，其加入对原序列的最长递增子序列无影响，因此，可得该序列的最长递增子序列为

3, 4, 5, 9

所以序列 A 的最长递增子序列的长度为 4。

注意：在解决最长递增子序列问题中，求解到的最长递增子序列可能不止一个。

令 $f(i)$表示序列中以 a_i 为末元素的最长递增子序列的长度，则在求以 a_i 为末元素的最长递增子序列时，找到所有序号在 i 前面且小于 a_i 的元素 a_j，即 $j<i$ 且 $a_j<a_i$。

如果这样的元素存在，那么对所有的 a_j，都有一个以 a_j 为末元素的最长递增子序列的长度 $f(j)$。把其中最大的 $f(j)$选出来，$f(i)$就等于最大的 $f(j)$加上 1，即以 a_i 为末元素的最长递增子序列，等于在使 $f(j)$最大的那个 a_j 为末元素的递增子序列最后再加上 a_i；如果这样的元素不存在，那么 a_i 自身构成一个长度为 1 的以 a_i 为末元素的递增子序列。

上述问题的代码如代码 8.4 所示。

代码 8.4　最长递增子序列的动态规划求解

```
def getdp1(arr):
    n = len(arr)
    dp = [0] * n
    for i in range(n):
        dp[i] = 1
        for j in range(i):
            if arr[i] > arr[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return dp
def generateLIS(arr, dp):
    n = max(dp)
    index = dp.index(n)
    lis = [0] * n
    n -= 1
    lis[n] = arr[index]
    # 从右向左
    for i in range(index, 0 - 1, -1):
        # 关键
        if arr[i] < arr[index] and dp[i] == dp[index] - 1:
            n -= 1
            lis[n] = arr[i]
            index = i
    return lis
```

该算法的时间复杂度为 $O(n^2)$，下面介绍可以将时间复杂度降为 $O(n\log n)$的改进算法。

8.4.2 改进算法

在基本算法中我们发现，当需要计算前 i 个元素的最长递增子序列时，前 i–1 个元素作为最大元素的各递增序列，无论是长度，还是最大元素值，都毫无规律可循，所以开始计算前 i 个元素的时候只能遍历前 i–1 个元素，以找到满足条件的 j 值，使得 $a_j < a_i$，且在所有满足条件的 j 中，以 a_j 作为最大元素的递增子序列最长。

在本节中，我们实现一个更高效的算法。

在一个序列中，长度为 n 的递增子序列可能不止一个，但是在所有长度为 n 的子序列中，有一个子序列是比较特殊的，那就是最大元素最小的递增子序列。如在 8.4.1 节的示例中，当处理到第 7 个元素“6”时，在当前已生成的子序列中，长度为 2 的子序列有两个，即序列{3, 4}和序列{1, 2}，则此时对长度为 2 的序列而言，{1, 2}就是最大元素最小的递增子序列。随着元素的不断加入，满足条件的子序列会不断变化。

如果将这些子序列按照长度由短到长排列，将它们的最大元素放在一起，形成新序列 $B\{b_1,b_2,\ldots,b_j\}$，则序列 B 满足 $b_1 < b_2 < \ldots < b_j$。

这个关系比较容易说明，假设 b_{xy} 表示序列 A 中长度为 x 的递增序列中的第 y 个元素，显然，如果在序列 B 中存在元素 $b_{mm} > b_{nn}$，且 $m < n$，则说明子序列 B_n 的最大元素小于序列 B_m 的最大元素，因为序列是严格递增的，所以在递增序列 B_n 中存在元素 $b_{nm} < b_{nn}$，且从 b_{n0} 到 b_{nm} 形成了一个新的长度为 m 的递增序列。

因为 $b_{mm} > b_{nn}$，所以 $b_{mm} > b_{nm}$，这就说明在序列 B 中还存在一个长度为 m，最大元素为 b_{nm} $< b_{mm}$ 的递增子序列，这与序列的定义，b_{mm} 是所有长度为 m 的递增序列中第 m 个元素最小的序列不符，所以序列 B 中的各元素严格递增。

发现了如此一个严格递增的序列，这让我们眼前一亮，可以利用此序列的严格递增性，使用二分查找，找到最大元素刚好小于 a_j 的元素 b_k，将 a_j 加入这个序列尾部，形成长度为 k+1 但是最大元素又小于 b_{k+1} 的新序列，取代之前的 b_{k+1}。如果 a_j 比 B_n 中的所有元素都要大，说明发现了以 a_j 为最大元素，长度为 n+1 的递增序列，将 a_j 作为 B_{n+1} 的第 n+1 个元素。从 b_1 依次递推，就可以在 $O(n\log n)$的时间内找出序列 A 的最长递增子序列。

理论说明比较枯燥，来看一个例子，以序列{6，7，8，9，10，1，2，3，4，5，6}来说明改进算法的步骤。

程序开始时，最长递增序列长度为 1（每个元素都是一个长度为 1 的递增序列），当处理第

2 个元素时发现 7 比最长递增序列 6 的最大元素还要大，所以将 6 和 7 结合生成长度为 2 的递增序列，说明已经发现了长度为 2 的递增序列，依次处理，到第 5 个元素“10”，这一过程中 *B* 数组的变化过程是

6

6，7

6，7，8

6，7，8，9

6，7，8，9，10

第 6 个元素是“1”，查找比“1”大的最小元素，发现是长度为 1 的子序列的最大元素 6，说明 1 是最大元素更小的长度为 1 的递增序列，用 1 替换 6，形成新序列(1，7，8，9，10)。然后查找比第 7 个元素“2”大的最小元素，发现是“7”，说明存在长度为 2 的序列，其末元素 2 比 7 更小，用 2 替换 7，依次执行，直到所有元素处理完毕，生成新的序列(1，2，3，4，5)，最后将“6”加入 *B* 数组，形成长度为 6 的最长递增子序列。

这一过程中，*B* 数组的变化过程是

1，7，8，9，10

1，2，8，9，10

1，2，3，9，10

1，2，3，4，10

1，2，3，4，5

1，2，3，4，5，6

当处理第 10 个元素“5”时，传统算法需要查看 9 个元素（6,7,8,9,10,1,2,3,4），而改进算法只需用二分查找法查找数组 *B* 中的两个元素（3, 4），可见效率可以大幅提升。

8.4.3　最终代码

最长递增子序列的动态规划求解改进算法的代码如代码 8.5 所示。

代码 8.5　最长递增子序列的动态规划求解改进算法

```
01  def getdp2(arr):
02      n = len(arr)
03      dp, ends = [0] * n, [0] * n
```

```
    ends[0], dp[0] = arr[0], 1
    right, l, r, m = 0, 0, 0, 0
    for i in range(1, n):
        l = 0
        r = right
        # 二分查找,若找不到则 ends[l 或 r]是比 arr[i]大而又最接近其的数
        # 若 arr[i]比 ends 有效区的值都大，则 l=right+1
        while l <= r:
            m = (l + r) / 2
            if arr[i] > ends[m]:
                l = m + 1
            else:
                r = m - 1
        right = max(right, l)
        ends[l] = arr[i]
        dp[i] = l + 1
    return dp
```

二分查找的结果，目的是找到这样一个 j，使满足 $A[B[j]] > A[i]$的所有 j 中，j 取得最小值。但是在二分查找的时候可能会发生两种特殊情况，B 数组中的所有元素都不小于 $A[i]$，B 数组中的所有元素都比 $A[i]$小，对于这两种情况需要专门处理。

- 对于 B 中所有元素都不小于 $A[i]$的情况，要将 $A[i]$更新到 $B[0]$的位置。
- 对于 B 中所有元素都小于 $A[i]$的情况，要将 $A[i]$更新到 $B[\max]$的位置，同时将 max 值增加，说明找到了比当前最长的递增序列更长的结果。

对于其他情况，在更新新节点的前驱节点时，要注意当前元素的前驱节点是 $B[j-1]$，而不是 $\text{pre}[B[j]]$，这一点要格外留意，后者看似有道理，但实际上在之前的更新中可能已经被变更过。

第 9 章

最短路径问题

你知道手机导航是怎么得出两个地点间的最短路线的吗？其实，手机把地图简化为一个加权图，然后利用这个加权图查找最短路线。把地点看成节点，把道路看成边，整个地图就可看成一个加权图。计算加权图中两点间的最短路径是编程的一个重要问题，这一章会用以下几个算法解决这个问题。

- 迪可斯特朗算法
- Floyd 算法
- A*算法

9.1 迪可斯特朗算法

小朗朗想坐公交车从 A 站到 F 站，如图 9.1 所示，但他看着下面的公交车线路图有些眼花，不知道怎么坐车距离最短。这时，他想到了迪可斯特朗算法。

迪可斯特朗算法（Dijkstra’s algorithm），又被译为戴克斯特拉算法，它主要用于解决有向图的最短路径问题。输入边的权重和一个起点，迪可斯特朗算法能够得出从该起点到任何其他节点的最短路径。从图中的某个顶点出发到达另外一个顶点所经过的边的权重和最小的一条路径，称为最短路径。

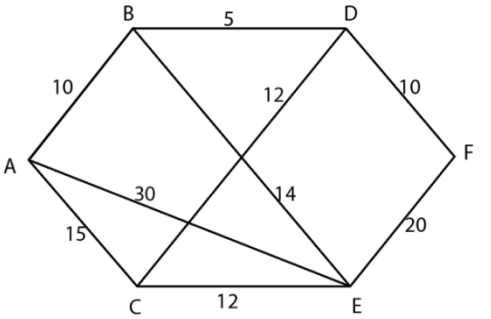


图9.1　公交车线路图

注意：权重不能为负。

它的发明者，艾兹格·迪可斯特朗（Edsger W. Dijkstra），是一位荷兰计算机科学家。他在二十分钟内，没有用纸和笔，只靠头脑就“想”出了现在的迪可斯特朗算法。

最初，迪可斯特朗想解决的问题是：如果我有荷兰 64 个城市的交通网络图，怎么样能让计算机找出任意两个城市之间的最短路径呢？20 分钟后，迪可斯特朗算法诞生，它完美地解决了这个问题。现在，迪可斯特朗算法被广泛地应用在地铁、公交车，以及飞机的路线查询系统中。

9.1.1　术语释义

如果你对图不是很熟悉的话，可以先通过表 9.1 了解一些关于图的基本术语。

表 9.1　术语释义

图	一个图是由节点和连接这些节点的边组成的
节点	一个小圆点，可以看成一个事物、站点、城市……
边	连接节点的直线或曲线
有向图	赋予边方向的图
权重	边的距离
前任节点	路线中连接节点的上一个节点

9.1.2　问题示例：最短公交线路

回到小朗朗的问题，先来观察一下上面的图 9.1。这个公交车系统包括 6 个站点（分别用字母 A、B、C、D、E 和 F 来表示）和连接这些车站的 9 条边。图中每条边上都标着距离，这里的距离就是权重。另外，每条边都是双向的，可以从一个车站到另一个车站，也可以从那一个车站回到这个车站。

迪可斯特朗算法的基本概念是，如果设定一个车站为起点，那么算法能够找出一个公共交通系统中从该车站到任意其他车站的最短距离路线。迪可斯特朗算法的主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。下面来分步介绍一下这个算法。

9.1.3　图与节点的定义

首先，看一下图的定义，一个图由节点和连接这些节点的边组成。

```
01  class Graph:
02    def   init  (self):
03      self.nodes = set()                                    #节点集合
04      self.edges = defaultdict(list)                        #存储边集合的字典
05      self.distances = {}                                   #记录每条边的距离的集合
06
07    def add node(self, node):                               #增加节点
08      self.nodes.add(node)
09
10    def add edge(self, from node, to node, distance):       #增加边，输入参数（起点、终点、距离）
11      self.edges[from node].append(to node)
12      self.distances[(from_node, to_node)] = distance
```

接下来看一下节点的定义。

```
01  class Node:
02
03   def __init__(self, name):                                #参数：节点名称
04      self.distance = sys.maxint                            #起始距离值为最大值
05      self.predecessor = None                               #起始点的前任节点为空
06      self.name = name
07
08   def set_distance(self, dist):                            #设定距离值
09      self.distance = dist
10
11   def set_predecessor(self, pred):                         #设定前任节点
12      self.predecessor = pred
13
14   def get_distance(self):                                  #获得距离值
15      return self.distance
16
17   def get_predecessor(self):                               #获得前任节点
18      return self.predecessor
```

对于每个节点，需要存储到目前为止从起点 A 到这个节点的最短距离（距离值）。另外，

除了距离值，每个节点还需要存储它的前任节点（predecessor），也就是目前最短距离路线中它之前的那个节点。只有这样，最后才能原路返回到起点。

9.1.4 把图用代码“画”出来

遵循以上定义，创建图 9.1 所示的实例。

```
g=Graph()                                    #创建图的实例
g.add_node(Node('A'))                        #添加节点
g.add_node(Node('B'))
g.add_node(Node('C'))
g.add_node(Node('D'))
g.add_node(Node('E'))
g.add_node(Node('F'))
g.add_edge('A', 'B', 10)                     #添加边：传入其起点、终点及距离
g.add_edge('B', 'A', 10)                     #因为是双向图，所以也需要添加反方向的边
g.add_edge('A', 'C', 15)
g.add_edge('C', 'A', 15)
…
g.add_edge('E', 'F', 20)
g.add_edge('F', 'E', 20)
```

9.1.5 算法核心：两个节点集合

接下来，需要定义两个节点集合：临时（temporary）节点集合和永久（permanent）节点集合。临时节点集合用于存储距离值和改变空间的节点；永久节点集合用于存储距离值已经固定的节点。在算法起始，临时集合只存放起点 A，永久集合则为空。来看一下代码：

```
permanent = set()                            #永久节点集合
temporary = set()                            #临时节点集合
temporary.add(initial)                       #起始时把起点加入临时节点集合
```

9.1.6 算法核心：循环

创建完图的实例、永久节点集合和临时节点集合后，开始运行算法。迪可斯特朗算法一直循环两个步骤，和贪心算法很像，它每循环一次都确保局部最优解。

步骤 1：找到临时节点集合中距离值最小的节点，把它迁移到永久集合里。

这么做的逻辑是什么？想一想，临时节点集合里距离值最小的节点肯定已经找到了最短路线，因为临时节点集合里任何其他节点的距离值都比它的大或者和它一样，所以如果改变路线的话它的最短距离值只会增大。因此，可以确认它的距离值不会再减少。

步骤 2：检查该节点的相邻节点集合。如果相邻节点（neighbour）不属于永久集合，则尝试更新其距离值，一旦更新成功便把该相邻节点加入临时节点集合。

循环步骤 1 与步骤 2，直至临时节点集合为空。当所有节点都被框起来时，循环终止。这代表所有节点都找到了最小距离值与最短路线。来看一下代码：

```
while temporary:                                          #循环直至临时节点集合为空
  min node = None
  for node in temporary:            #用 for 循环找到临时节点集合里距离值最小的节点 min_node
     if min node is None:
       min_node = node
     elif node.get_distance() < min_node.get_distance():
       min_node = node
  temporary.remove(min node)        #把 min node 从临时节点集合迁移到永久集合
  permanent.add(min_node)
   current_distance = min_node.get_distance()          #当前节点（min_node）的距离值
  for neighbour in graph.edges[min_node]:              #检查当前节点的每个相邻节点
    new distance = current distance + graph.distance[(min node, neighbour)]
                                    #新的距离值=当前节点（min_node）的距离值+边的距离值
    if neighbour not in permanent and new_distance < neighbour.get_distance():
                                    #如果新的距离值小于相邻节点的距离值
      neighbour.set_distance(new_distance)             #更新相邻节点的距离值
      neighbour.set_predecessor(min_node)              #更新相邻节点的前任
      temporary.add(neighbour)                         #把相邻节点加入临时节点集合
```

9.1.7　输出路线

从终点回溯到起点就可以得到最短路线，这也是程序的最后一步。我们想要的输出结果是一个节点集合，节点以最短路线中的顺序展现。以下代码展示了这一步骤：

```
def printPath(self,end,predecessor):      #输出路线方法
  current = end
  path = {end}                             #路线节点集合
  while current.predecessor != None        #只要当前节点有前任节点
           path.add(current.predecessor)   #把前任节点加入集合
           current = current.predecessor   #更新当前节点
  path.reverse()                           #翻转集合令终点为最后一个元素，起点为第一个元素
  print path                               #输出集合
```

9.1.8　通过示例理解算法

如果你觉得看代码不是很明白，下面在示例图上模拟一遍思路，这样你就会对这个算法有一个比较透彻的理解。如图 9.2 所示，用黑框代表存放在永久集合里的节点。每一次循环，都

需要找到临时节点集合里距离值最小的节点，并把它迁移到永久集合里（用黑框框起来）。因为在算法起始，只有起点 A 在临时节点集合里，所以起点 A 就是第一次循环时临时节点集合里距离值最小的节点。

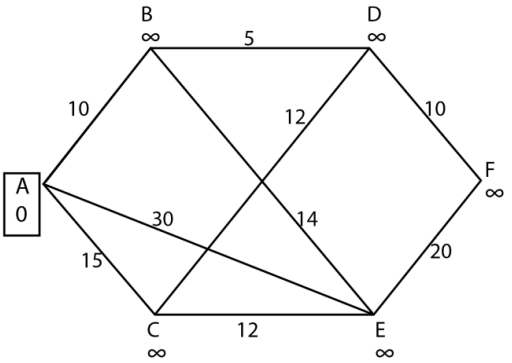


图9.2　第一次循环

接着，检查节点 A 的相邻节点集合{B,C,E}。记住，如果相邻节点属于永久集合，则跳过；如果相邻节点不属于永久集合，则尝试更新其距离值，一旦更新成功就把该相邻节点加入临时节点集合。

A 的相邻节点 B、C 和 E 都不属于永久集合，所以可以尝试更新其距离值。如果经过 A 的路线距离值比原本距离值小，则更新距离值。更新距离值的过程有一个专业术语叫作“松弛”。这里拿节点 B 来举例。节点 B 的原本距离值理论上无限大，为最大值，经过 A 的话，B 的距离值是 10，所以更新它的距离值，并且设定其前任节点为节点 A。另外，把节点 B 加入临时节点集合。类似的，如图 9.3 所示，更新 C 和 E 的距离值、前任节点，并把它们加入临时节点集合。

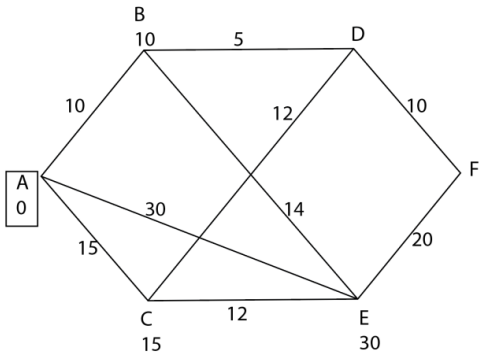


图9.3　更新B、C、E的距离值

现在，临时节点集合为{B,C,E}，因为临时节点集合不为空，所以循环步骤 1 与步骤 2。此

时，临时节点集合中距离值最小的节点为 B，所以把它迁移到永久集合里，如图 9.4 所示。

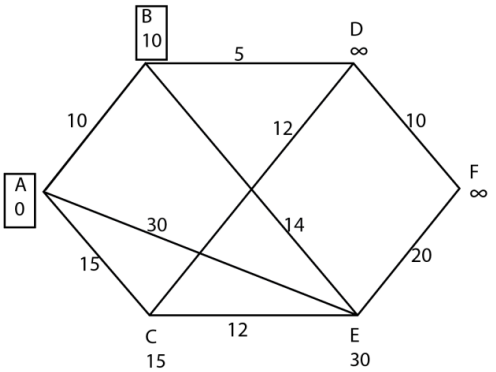


图9.4　第二次循环

重复第一次循环时的步骤，检查节点 B 的相邻节点集合{A,D,E}。其中，A 属于永久集合，不属于永久集合的相邻节点是 D 和 E。对比新老距离值，应该更新节点 D 和节点 E 的距离值，并设定它们的前任节点为节点 B，如图 9.5 所示。

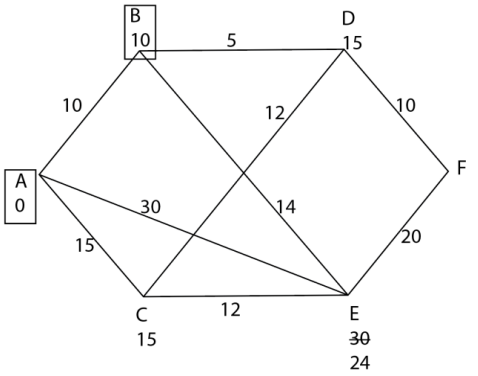


图9.5　更新D和E的距离值

专业一点说，A 节点到 D 节点的路程，通过边 AB 松弛成功；A 节点到 E 节点的路程，通过边 AE 松弛成功。这便是迪可斯特朗算法的主要思想：通过“边”来松弛起始节点到其余各个节点的路程。接着，把节点 D 和节点 E 加入临时节点集合。

注意：虽然节点 E 已经在临时节点集合中，不过集合的特性是不复制。

就这样，在临时节点集合不为空的情况下，一直循环步骤 1 与步骤 2。当所有节点都被框起来时，循环终止。那么，小朗朗想得到从 A 到 F 的最短距离路线。通过前任节点集合，他知道 F 站之前是 D 站，D 站之前是 B 站，B 站之前是 A 站。所以反过来，小朗朗得到的最短路线

是 A—B—D—F，如图 9.6 所示。

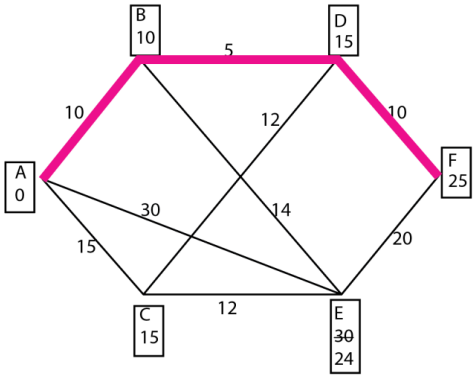


图9.6　最终输出的最短距离路线

9.1.9　完整代码展示

最短距离路线算法的完整代码如代码 9.1 所示。

代码 9.1　利用迪可斯特朗算法解决图的最短路线问题

```
class Graph:                                              #Graph 类

 def __init__(self):
      self.nodes = set()                                  #节点集合
      self.edges = defaultdict(list)                      #边集合
      self.distances = {}                                 #边的距离的集合

 def add_node(self, value):                               #添加节点
      self.nodes.add(value)

 def add_edge(self, from_node, to_node, distance):        #添加边
      self.edges[from node].append(to node)
      self.distances[(from node, to node)] = distance

class Node:                                               #节点类

 def   init  (self, name):
      self.distance = sys.maxint                          #距离值
      self.predecessor = None                             #前任节点
      self.name = name                                    #名称

 def set_distance(self, dist):                            #设距离值
```

```
23        self.distance = dist
24
25    def set predecessor(self, pred):                          #设前任节点
26        self.predecessor = pred
27
28    def get distance(self):                                   #得到距离值
29        return self.distance
30
31    def get predecessor(self):                                #得到前任节点
32        return self.predecessor
33
34def dijsktra(graph, initial, end):
35
36    permanent = {}                                            #永久集合
37    temporary = {}                                            #临时集合
38    temporary.add(initial)                                    #将起点加入临时集合
39    initial.set distance(0)                                   #设起点距离值为 0
40
41    while temporary:                                          #只要临时集合不为空
42        min_node = None
43        for node in temporary:                                #找临时集合中距离值最小的节点
44            if min_node is None:
45                min_node = node
46            elif node.get_distance() < min_node.get_distance():
47                min_node = node
48        temporary.remove(min_node)                            #把该节点从临时集合中移除
49        permanent.add(min_node)                               #将节点加到永久集合
50        current_distance = min_node.get_distance()
51        for neighbour in graph.edges[min_node]:               #遍历相邻节点
52            new_distance = current_distance + graph.distance[(min_node, neighbour)]
53            if neighbour not in permanent and new_distance < neighbour.get_distance():
                                          #如果相邻节点不在永久集合中,且新距离值小于原距离值
54                neighbour.set_distance(new_distance)          #更新
55                neighbour.set_predecessor(min_node)
56                temporary.add(neighbour)
57 def printPath(self,end,predecessor):                        #输出路线方法
58    current = end
59    path = {end}                                              #路线节点集合
60    while current.predecessor != None                         #只要当前节点有前任节点
61        path.add(current.predecessor)                         #把前任节点加入集合
62        current = current.predecessor                         #更新当前节点
63    path.reverse()                                            #翻转集合
64    print path                                                #输出集合
```

9.2 Floyd 算法

Floyd 算法和迪可斯特朗算法一样，可用于解决加权图的最短路径问题。Floyd 算法是罗伯特·弗洛伊德发明的。弗洛伊德是图灵奖得主，但他并没有受过正规的计算机教育，是自学成才的。

小朗朗还是有同样的问题：他想坐公交车从 A 站到 F 站，但他看着图 9.7 所示的公交车线路图头疼，不知道怎样坐车距离最短。上一节讲述了怎么用迪可斯特朗算法得出从 A 站到 F 站的最短路线，下面让我们用 Floyd 算法解决这个问题。

注意：迪可斯特朗算法要求权重不为负数，Floyd 算法则没有这项要求。

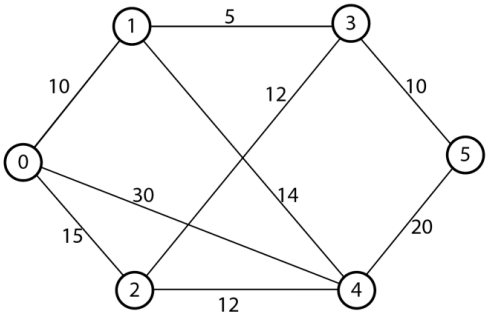


图9.7　公交车线路图

9.2.1 算法核心：两个矩阵

Floyd 算法比迪可斯特朗算法更加简单易懂。需要创建两个矩阵：距离 ***D*** 矩阵和路线 ***P*** 矩阵。***D*** 矩阵[*i*][*j*]中的值代表从 *i* 节点到 *j* 节点的最短距离。***P*** 矩阵[*i*][*j*]中的值代表从 *i* 节点到 *j* 节点的最短路径中经过的第一个节点。看表 9.2 和表 9.3，这是初始化时的 ***D*** 矩阵和 ***P*** 矩阵。

表 9.2　初始化时的 *D* 矩阵

D	0	1	2	3	4	5
0	0	10	15	MAX	30	MAX
1	10	0	MAX	5	14	MAX
2	15	MAX	0	12	12	MAX
3	MAX	5	12	0	MAX	10
4	30	14	12	MAX	0	20
5	MAX	MAX	MAX	10	20	0

D 矩阵记录着加权图中边的权重（距离），如果两个节点没有边相连，在 ***D*** 矩阵中用 MAX 记录。注意，你会发现 ***D*** 矩阵是关于对角线对称的，因为距离[*i*][*j*]等于距离[*j*][*i*]。

表 9.3　初始化时的 *P* 矩阵

P	0	1	2	3	4	5
0	0	1	2	3	4	5
1	0	1	2	3	4	5
2	0	1	2	3	4	5
3	0	1	2	3	4	5
4	0	1	2	3	4	5
5	0	1	2	3	4	5

P 矩阵记录从 *i* 节点到 *j* 节点的最短路线中经过的第一个点。比如，从 1 节点到 4 节点用 ***P***[1][4]表示。开始的时候，从 *i* 到 *j* 的路线是直达的，所以经过的第一个节点就是 *j* 节点。

D 矩阵需作为参数输入，***P*** 矩阵则根据节点数量自动生成。以下代码生成 ***P*** 矩阵：

```
01 def createPath(self,n):                    #创建 P 矩阵
02       path=[]                              #一个长度为 n，由[0,1,2,3,…,n-1]构成的二维数组
03       for i in range(n):
04          row = []
05          for j in range(n):
06             row.append(j)
07          path.append(row)
08       return path
```

9.2.2　算法核心：通过中介点缩短距离

依次令图中的节点为 *k* 节点，并使 *k* 节点作为中介点，如果 ***D***[*i*][*k*]+***D***[*k*][*j*]<***D***[*i*][*j*]，也就是说，如果由中介点到达 *j* 点的距离比直接从 *i* 到 *j* 的距离短的话，就更新 ***D*** 矩阵和 ***P*** 矩阵：***D***[*i*][*j*]= ***D***[*i*][*k*]+***D***[*k*][*j*]、***P***[*i*][*j*]= ***P***[*i*][*k*]，缩短 ***D*** 矩阵的最短距离值，并更改 ***P*** 矩阵的最短距离路线。

现在来看一下核心代码。solveFloyd 方法有以下参数：

- dist，距离矩阵。传入的矩阵记录是节点之间边的距离，如果两个节点间没有边，则记录为 sys.maxsize。
- start，起始节点。
- end，终止节点。

```
01 def solveFloyd(self,dist,start,end):
02       n=len(dist)                      #n 为节点数量
03       path = self.createPath(n)        #创建 P 矩阵
04       for k in range (n):              #依次设节点为中介点 k
05          for i in range(n):            #依次设节点为起点 i
06             for j in range(n):         #依次设节点为终点 j
07                if dist[i][j] > dist[i][k] + dist[k][j]:   #如果通过中介点路程更短
08                   dist[i][j] = dist[i][k] + dist[k][j]    #更新 D 矩阵
09                   path[i][j] = path[i][k]                 #更新 P 矩阵
10       self.printPath(start,path,end) #输出 start 节点到 end 节点的最短路线
```

9.2.3 通过示例理解算法

下面把前面所讲的解题思路用示例过一遍，以便更好地理解 Floyd 算法。如图 9.8 所示，依次使节点 0~5 为中介点 *k*。首先，从 0 节点开始。

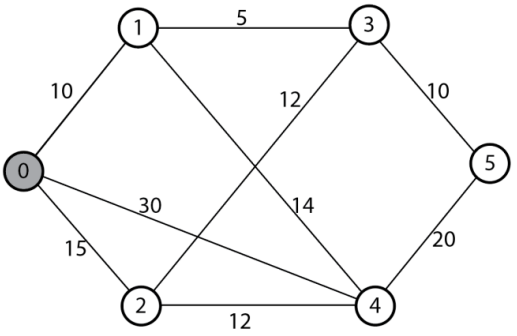


图9.8　中介点*k*=节点0

以 0 节点为中介点，从 1 节点到 2 节点的距离由 MAX 缩短为 25，因此，更新 ***D*** 矩阵的[1][2]和[2][1]值为 25，如表 9.4 所示。

表 9.4　0 节点为中介点，更新 ***D*** 矩阵

D	0	1	2	3	4	5
0	0	10	15	MAX	30	MAX
1	10	0	**25**	5	14	MAX
2	15	**25**	0	12	12	MAX
3	MAX	5	12	0	MAX	10
4	30	14	12	MAX	0	20
5	MAX	MAX	MAX	10	20	0

另外，更新 ***P*** 矩阵，如表 9.5 所示。从 1 节点到 2 节点的最短路线不再是 1—2，而是 1—0—2，因此 ***P***[1][2]等于 0。同样，从 2 节点到 1 节点的最短路线不再是 2—1，而是 2—0—1。因此 ***P***[2][1]等于 0。注意，***P***[*i*][*j*]不一定等于 ***P***[*j*][*i*]，两条路线经过的第一个节点有可能不相同。

表 9.5　0 节点为中介点，更新 *P* 矩阵

P	0	1	2	3	4	5
0	0	1	2	3	4	5
1	0	1	**0**	3	4	5
2	0	**0**	2	3	4	5
3	0	1	2	3	4	5
4	0	1	2	3	4	5
5	0	1	2	3	4	5

接下来以 1 节点为中介点，如图 9.9 所示。0 节点到 3 节点的距离从 MAX 缩短为 15，0 节点到 4 节点的距离从 30 缩短为 24，3 节点到 4 节点的距离从 MAX 缩短为 19。

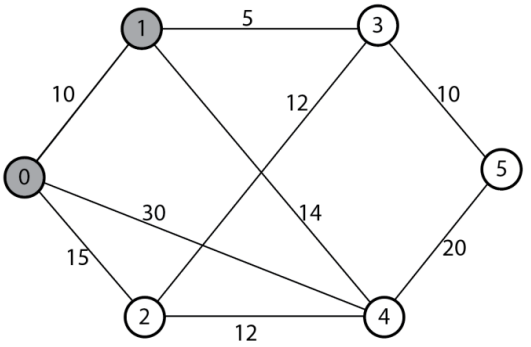


图9.9　中介点k=节点1

如表 9.6 所示，更新 ***D*** 矩阵，***D***[0][3]和 ***D***[3][0]从 MAX 变为 15，***D***[0][4]和 ***D***[4][0]从 30 变为 24，***D***[3][4]和 ***D***[4][3]从 MAX 变为 19。

表 9.6　1 节点为中介点，更新 *D* 矩阵

D	0	1	2	3	4	5
0	0	10	15	**15**	**24**	MAX
1	10	0	**25**	5	14	MAX
2	15	**25**	0	12	12	MAX
3	**15**	5	12	0	**19**	10

续表

D	0	1	2	3	4	5
4	**24**	14	12	**19**	0	20
5	MAX	MAX	MAX	10	20	0

如表 9.7 所示，更新 ***P*** 矩阵。从 0 节点到 3 节点的最短路线从 0—3 变为 0—1—3，所以 ***P***[0][3]等于 1。从 0 节点到 4 节点的最短路线从 0—4 变为 0—1—4，所以 ***P***[0][4]等于 1。同理，***P***[3][0]、***P***[4][0]、***P***[3][4]和 ***P***[4][3]也等于 1。

表 9.7　1 节点为中介点，更新 *P* 矩阵

P	0	1	2	3	4	5
0	0	1	2	**1**	**1**	5
1	0	1	**0**	3	4	5
2	0	**0**	2	3	4	5
3	**1**	1	2	3	**1**	5
4	**1**	1	2	**1**	4	5
5	0	1	2	3	4	5

以 2 节点为中介点不改变任何路线，直接以 3 节点为中介点，如图 9.10 所示。从 0 节点到 5 节点的距离由 MAX 变为 25，从 1 节点到 2 节点的距离由 25 变为 17，从 1 节点到 5 节点的距离由 MAX 变为 15，从 2 节点到 5 节点的距离由 MAX 变为 22。

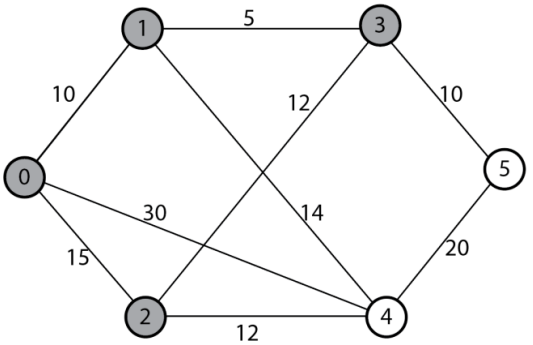


图9.10　中介点k=节点3

如表 9.8 所示，更新 ***D*** 矩阵。***D***[0][5]和 ***D***[5][0]等于 25，***D***[1][2]和 ***D***[2][1]等于 17，***D***[1][5]和 ***D***[5][1]等于 15，***D***[2][5]和 ***D***[5][2]等于 22。

表 9.8　3 节点为中介点，更新 *D* 矩阵

D	0	1	2	3	4	5
0	0	10	15	**15**	**24**	**25**
1	10	0	**17**	5	14	**15**
2	15	**17**	0	12	12	**22**
3	**15**	5	12	0	**19**	10
4	**24**	14	12	**19**	0	20
5	**25**	**15**	**22**	10	20	0

如表 9.9 所示，更新 ***P*** 矩阵。从 0 到 5 的路线从 0—5 变为 0—1—3—5，因此 ***P***[0][5]等于 1。从 5 到 1 的路线从 5—1 变为 5—3—1—0，因此 ***P***[5][0]等于 1。从 1 到 2 的路线从 1—2 变为 1—3—2，因此 ***P***[1][2]等于 3。同理，***P***[2][1]、***P***[1][5]、***P***[5][1]、***P***[2][5]、***P***[5][2]等于 3。

表 9.9　3 节点为中介点，更新 *P* 矩阵

P	0	1	2	3	4	5
0	0	1	2	**1**	**1**	**1**
1	0	1	**3**	3	4	**3**
2	0	**3**	2	3	4	**3**
3	**1**	1	2	3	**1**	5
4	**1**	1	2	**1**	4	5
5	**3**	**3**	**3**	3	4	5

以 4 节点和 5 节点为中介点不改变任何路线，略过它们。如图 9.11 所示，来看一下最初的 ***D*** 矩阵和最后所有值都是最短距离的 ***D*** 矩阵的对比。

D	0	1	2	3	4	5
0	0	10	15	MAX	30	MAX
1	10	0	MAX	5	14	MAX
2	15	MAX	0	12	12	MAX
3	MAX	5	12	0	MAX	10
4	30	14	12	MAX	0	20
5	MAX	MAX	MAX	10	20	0

D	0	1	2	3	4	5
0	0	10	15	15	24	25
1	10	0	17	5	14	15
2	15	17	0	12	12	22
3	15	5	12	0	19	10
4	24	14	12	19	0	20
5	25	15	22	10	20	0

图9.11　***D***矩阵对比图

来看一下 ***P*** 矩阵的对比图，如图 9.12 所示。

P	0	1	2	3	4	5
0	0	1	2	3	4	5
1	0	1	2	3	4	5
2	0	1	2	3	4	5
3	0	1	2	3	4	5
4	0	1	2	3	4	5
5	0	1	2	3	4	5

P	0	1	2	3	4	5
0	0	1	2	1	1	1
1	0	1	2	3	4	5
2	0	1	2	3	4	5
3	0	1	2	3	4	5
4	0	1	2	3	4	5
5	0	1	2	3	4	5

图9.12　***P***矩阵对比图

如果想知道从 0 节点到 5 节点的最短距离，只需要查看 ***D***[0][5]就可得知，这里，***D***[0][5]等于 25。如果想得出 0 节点到 5 节点的最短路线，只要查看 ***P*** 矩阵的 ***P***[0][5]便可得知路线的第一节点是 1。接着，查看 ***P***[1][5]便可得知路线的第二个节点为 3，第三个节点 ***P***[3][5]为 5，也就是终点。因此，得出的最短路线为 0—1—3—5。

9.2.4　完整代码

代码 9.2 所示的是 Floyd 算法的完整代码。

代码 9.2　利用 Floyd 算法解决图的最短路线问题

```
01 import sys
02 class Solution():
03   def solveFloyd(self,dist,start,end):   #核心方法
04      n=len(dist)
05      path = self.createPath(n)
06      for k in range (n):                 #遍历节点，设成中介点
07         for i in range(n):
08            for j in range(n):
09               if dist[i][j] > dist[i][k] + dist[k][j]:        #如果路程更短
10                  dist[i][j] = dist[i][k] + dist[k][j]         #更新 D、P 矩阵
11                  path[i][j] = path[i][k]
12      self.printPath(start,path,end)      #输出
13   def createPath(self,n):                #创建 P 矩阵
14      path=[]                             #一个长度为 n，由[0,1,2,3,…,n-1]构成的二维数组
15      for i in range(n):
16         row = []
17         for j in range(n):
```

```
            row.append(j)
        path.append(row)
    return path
 def printPath(self,current,path,end):  #输出方法
    solution = []                       #solution 为最短路线列表
    while current!=end:                 #只要当前节点不为终点
        solution.append(current)        #把当前节点加入 solution
        current = path[current][end]    #设当前节点为路线中的下一个节点
    solution.append(current)            #最后加入终点
    print solution                      #输出 solution
```

让我们试一试代码能不能解决一开始的问题：

```
Solution().solveFloyd([   [0,  10,  15,   sys.maxsize,   30,    sys.maxsize],
                          [10, 0,   sys.maxsize,    5,    14,    sys.maxsize],
                          [15, sys.maxsize,  0,    12,   12,     sys.maxsize],
                          [sys.maxsize,5,   12,     0,    sys.maxsize,   10],
                          [30,     14,      12,   sys.maxsize,   0,    20],
                          [sys.maxsize,sys.maxsize,sys.maxsize,  10,   20, 0]],0,5)
```

下面是输出结果：

```
[0, 1, 3, 5]
```

9.3　A*算法

A*算法，也称 A 星算法，由迪可斯特朗算法演变而来，用于寻求有权图中的最短路径。A*算法与迪可斯特朗算法的区别在于，A*算法通过评估函数来预估完整路线的距离，而迪可斯特朗算法只关心当前距离，而并非完整路线的距离。因为利用评估函数，可减少遍历的节点数量，从而可更快地到达终点。输入边的权重和一个起点，A*算法能够比迪可斯特朗算法更快地得出从该起点到任何其他节点的最短路径。评估函数输出的距离估算值与实际值的接近程度决定算法的最终搜索速度。

9.3.1　算法核心：迪可斯特朗算法

在学习 A*算法之前，请先理解迪可斯特朗算法。这两个算法的结构和逻辑几乎一模一样，只不过 A*算法多了一个预估函数。其实，迪可斯特朗算法可视为 A*算法的一个特例，其预估函数为 0。

简单概括一下迪可斯特朗算法。定义两个节点集合：临时节点集合和永久节点集合。临时节点集合用于存储距离值和改变空间的节点；永久节点集合用于存储距离值已经固定的节点。

在算法起始，临时集合只存放起点 A，永久集合则为空。接下来循环两个步骤直到临时节点集合为空：首先，找到临时节点集合中距离值最小的节点，把它迁移到永久集合里；其次，检查该节点的相邻节点集合。如果相邻节点不属于永久集合，则尝试更新其距离值，一旦更新成功便把该相邻节点加入临时节点集合。最后，从终点节点回溯到起点就可以得到最短路线。

再来看一下迪可斯特朗算法的核心代码。

```
    permanent = set()                                          #永久节点集合
    temporary = set()                                          #临时节点集合
    temporary.add(initial)                                     #起始时把起点加入临时节点集合
while temporary:                                               #循环直至临时节点集合为空
    min node = None
    for node in temporary:              #用 for 循环找到临时节点集合里距离值最小的节点 min node
        if min node is None:
          min node = node
        elif node.get distance() < min node.get distance():
          min node = node
    temporary.remove(min node)          #把 min node 从临时节点集合迁移到永久集合
    permanent.add(min node)
     current distance = min node.get distance()                #当前节点（min node）的距离值
    for neighbour in graph.edges[min node]:                    #检查当前节点的每个相邻节点
      new distance = current distance + graph.distance[(min node, neighbour)]
                                        #新的距离值=当前节点（min node）的距离值+边的距离值
      if neighbour not in permanent and new distance < neighbour.get distance():
                                        #如果新的距离值小于相邻节点的距离值
        neighbour.set distance(new distance)                   #更新相邻节点的距离值
        neighbour.set predecessor(min node)                    #更新相邻节点的前任
         temporary.add(neighbour)                              #把相邻节点加入临时节点集合
def printPath(self,end,predecessor):                           #输出路线方法
  current = end
  path = {end}                                                 #路线节点集合
  while current.predecessor != None                            #只要当前节点有前任节点
           path.add(current.predecessor)                       #把前任节点加入集合
           current = current.predecessor                       #更新当前节点
  path.reverse()                                               #翻转集合
  print path                                                   #输出集合
```

9.3.2 算法核心：预估函数

A*算法利用一个预估函数来估算完整路线的距离，这个预估函数也是 A*算法与迪可斯特朗算法的不同之处。让我们用变量 d 来表示当前距离（从起点到当前节点的距离），变量 h 表示预估剩余距离（当前节点到终点的距离），变量 f 表示预估完整路线距离，也就是 $d+h$。在迪可

斯特朗算法中，所谓的距离值指的是变量 *d*，而在 A*算法中，指的是变量 *f*。

变量 *d* 是当前路线经过的边的权重总和，变量 *h* 是用预估函数得出的一个数值。预估函数随着算法用途改变，但一个必要条件是输出数值必须小于实际数值。预估函数的精确性决定 A*算法的表现，优先最贴近实际数值的预估函数。

代码几乎没有变化，只需将所有当前完整距离改为预估完整路线距离。

```
01    permanent = set()                                    #永久节点集合
02    temporary = set()                                    #临时节点集合
03    temporary.add(initial)                               #起始时把起点加入临时节点集合
04  while temporary:                                       #循环直至临时节点集合为空
05    min node = None
06    for node in temporary:              #用 for 循环找到临时节点集合里距离值最小的节点 min_node
07       if min_node is None:
08         min node = node
09        elif (node.get_distance()+heuristic(node,end)) < (min_node.get_distance() +
heuristic (min_node,end):                                  #比较预估完整路线距离
10         min_node = node
11    temporary.remove(min_node)          #把 min_node 从临时节点集合迁移到永久集合里
12    permanent.add(min_node)
13     current_distance = min_node.get_distance() + heuristic (min_node,end)
                                          #当前节点（min_node）的距离值
14    for neighbour in graph.edges[min_node]:              #检查当前节点的每个相邻节点
15      new_distance = current_distance + graph.distance[(min_node, neighbour)] +
heuristic (neighbour,end)
16        #新的距离值=当前节点（min_node）的距离值+边的距离值+相邻节点的预估剩余距离
17      if neighbour not in permanent and new_distance < (neighbour.get_distance() +
heuristic (neighbour,end)):
18                                                  #如果新的预估距离值小于相邻节点的预估距离值
19        neighbour.set_distance(new_distance)      #更新相邻节点的距离值
20        neighbour.set_predecessor(min_node)       #更新相邻节点的前任
21         temporary.add(neighbour)                 #把相邻节点加入临时节点集合
22 def printPath(self,end,predecessor):             #输出路线方法
23  current = end
24  path = {end}                                    #路线节点集合
25  while current.predecessor != None               #只要当前节点有前任节点
26           path.add(current.predecessor)          #把前任节点加入集合
27           current = current.predecessor          #更新当前节点
28  path.reverse()                                  #翻转集合
29  print path                                      #输出集合
31 def heuristic(self,node,end):
32   …
```

9.3.3 算法核心：选择预估函数

以上没有定义预估函数 heuristic()，这是因为它的定义会根据算法的用途而改变。例如，寻找城市之间的最短路线，可以定义预估函数为两个城市之间的直线距离。在允许 4 个或 6 个方向操作的几何路网中，常定义预估函数为曼哈顿距离。在允许 8 个方向操作的几何路网中，常定义预估函数为对角线距离。不过，不论怎样定义预估函数，必须确保估算值比实际值小。如果估算值比实际值大，有可能漏掉正确路线。之前说过，预估函数越精确越好，也就是说，用来计算估算值时的约束条件越多越好。但是，约束条件越多程序就会花越多的时间进行计算，这个问题在游戏开发中需要注意，预估函数应该在精确度和计算量中寻找平衡。

9.3.4 A*算法的兄弟们

严格来说，我们一直说的 A*算法应该被称为 A 算法。如果预估函数的估算值偶尔比实际值大，算法称为 A*算法；如果估算值永远比实际值小，算法称为 A 算法；如果估算值永远等于实际值，还是称之为 A 算法。最后，如果预估函数的估算值永远为 0，称这个算法为迪可斯特朗算法。第三种可能不常见，因为很少能够做出完美的估算。为了方便，编程界将 A*算法与 A 算法统称为 A*算法。

如果预估函数的估算值偶尔比实际值大，有可能漏掉准确路线，不过没有关系，有的时候我们更关心时间复杂度。比起花较长的时间得到“完美的”路线，我们宁可花较少的时间得到一条“还可以”的路线。

第 10 章

分治算法

分治算法的核心思想是将一个规模很大的问题化简为 n 个规模较小的问题，这些子问题虽然独立而不同，但是问题的本质是一致的，从而达到分而治之的目的。

本章主要涉及的问题有：

- 归并排序
- 连续子列表的最大和
- 几何问题之凸包
- 数学问题之多项式乘法

10.1 什么是分治

分治算法可以用四个字概括：分而治之，原则为通过解决子问题得到总问题的解。如图 10.1 所示，首先把问题分解成 n 个子问题，再把子问题进一步分解，直到达到最小的子问题。这时，“治”子问题并利用子问题的解推导总问题的解。

子问题应与原问题拥有同样的结构，或者拥有同样的形式。只有这样，才能利用递归解决子问题。

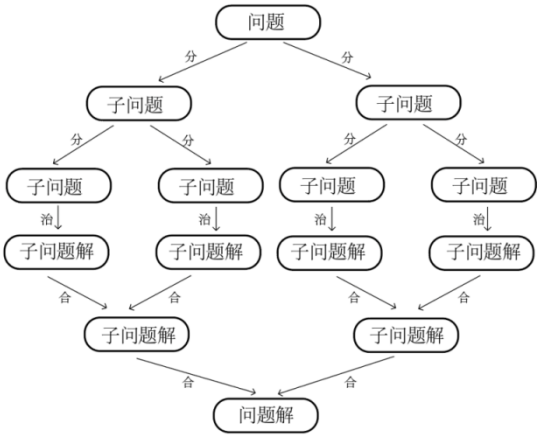


图10.1　分治算法

10.2　归并排序

归并排序是一种有效的排序算法。其他常见的排序算法包括冒泡排序、插入排序、二叉树排序、快速排序、堆排序等。归并排序是分治的一个非常经典的应用。

10.2.1　递归法与迭代法

归并算法有两种类型：递归法（Top-down）和迭代法（Bottom-up）。

递归法的核心思想是把列表分解成两个子列表，单独排序子列表再合并子列表。如图 10.2 所示，递归法可以利用二叉树理解：列表为根节点，子列表为子节点。首先排序子列表，再一步步合并子列表。这个算法之所以被称为递归法是因为它利用递归的思想，把根节点的问题递给子节点，一环套一环地解决问题。

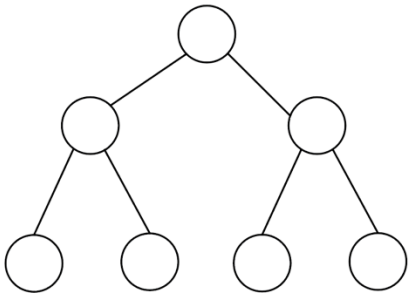


图10.2　递归法展现

第二种归并算法是迭代法，即利用迭代的方法排序。还是想排序一个有 n 个元素的列表，与其把列表看成一个整体，按照迭代法的想法，不如一开始就把列表看成 n 个长度为 1 的子列表。然后，利用循环把相邻的两个子列表排序合并，得到 $n/2$ 个长度为 2 的子列表。之后，再次利用循环把相邻的两个子列表排序合并，得到 $n/4$ 个长度为 4 的子列表，依此类推。最后，得到一个长度为 n 的完整列表。如图 10.3 所示，可以把迭代法的步骤想象成一个倒二叉树，对独立子列表进行多次迭代，直到到达根。

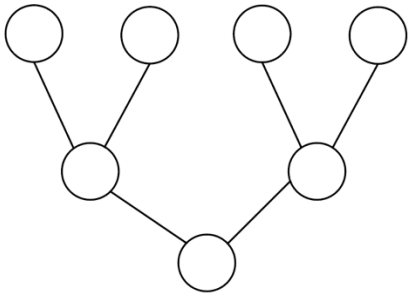


图10.3　迭代法展现

虽然归并排序有两种不同的实现方式，但两种方法的核心思想都是分治，也就是说，先解决小问题，再解决大问题。在这个排序问题中，先合并排序两个元素，再合并排序四个元素，再合并排序八个元素，依此类推。

10.2.2　递归法描述

递归法的主要思想是：把列表从中间分成两个子列表，把两个子列表归并排序，再把两个子列表合并。怎么把子列表归并排序呢？下面用图 10.4 中的例子解释归并排序。

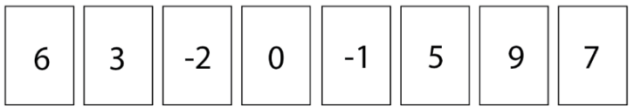


图10.4　列表

首先，如图 10.5 所示，把列表一分为二。

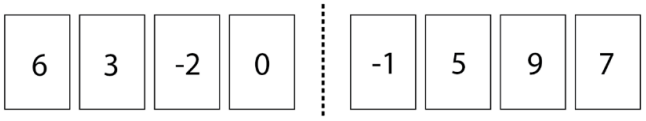


图10.5　第一步

怎么把子列表归并排序呢？先来看左子列表，按照算法，应该把左子列表也一分为二，如

图 10.6 所示。现在的任务是把这个左子列表的两个子列表归并排序。

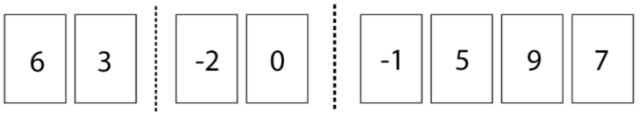


图10.6　第二步

先来看左子列表的左子列表，还是按照算法把列表一分为二，如图 10.7 所示。

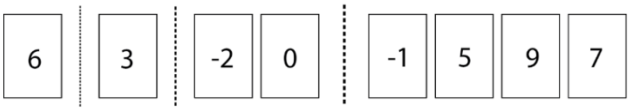


图10.7　第三步

现在有两个长度为 1 的列表，[6]与[3]。这两个列表不用排序，因为它们只有一个元素。下一步是把这两个列表合并，如图 10.8 所示，定义两个指针：A 指针与 B 指针。A 指向第一个列表的第一个元素，B 指向第二个列表的第一个元素。如果 A 指着的元素大，就把这个元素放在第一个位置，否则，将 B 指着的元素放在第一个位置。

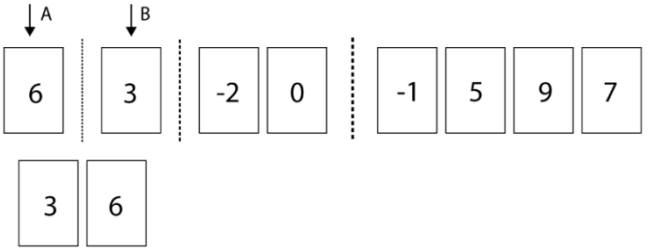


图10.8　第四步

以同样的方式对左子列表的右子列表排序，如图 10.9 所示。

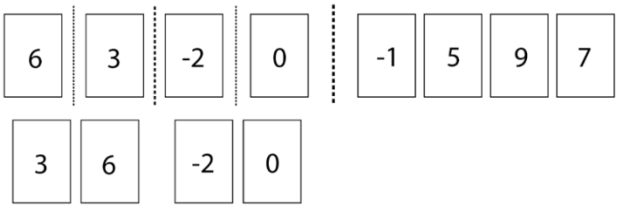


图10.9　第五步

现在的任务是把这两个排好序的子列表合并。如图 10.10 所示，还是用指针的方法，定义指针 A 与指针 B。比较 3 与−2，−2 小所以把−2 排在第一位。移动指针 B，比较 3 与 0，0 小所

以把 0 放在第二位，指针 B 已经走到头，所以 3 与 6 肯定是第三位与第四位。如图 10.11 所示，得到排好序的完整列表的左子列表。

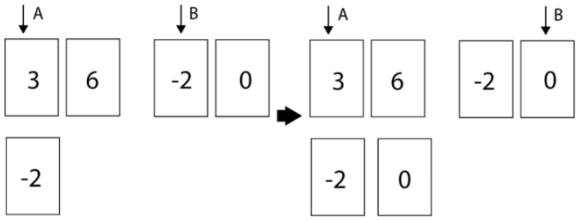


图10.10　第六步

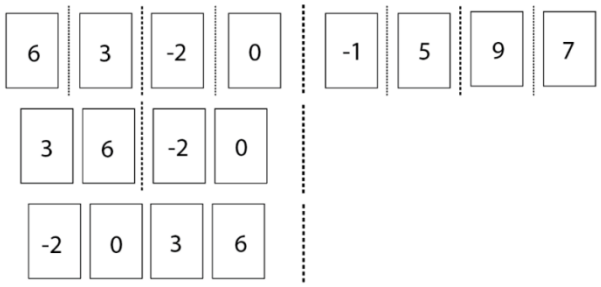


图10.11　第六步结果

把列表的左子列表排序完，才能开始排序列表的右子列表。排序过程是完全一样的，操作步骤就不一一展示，直接来看一下结果吧。如图 10.12 所示，列表的左子列表和右子列表排序完毕后得到[−2, 0, 3, 6] 与[−1, 5, 7, 9]。用指针的方法，合并两个列表，得到最终排序完毕的完整列表[−2, −1, 0, 3, 5, 6, 7, 9]。

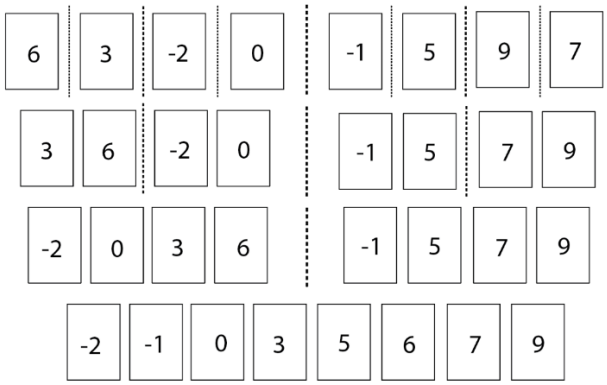


图10.12　完整列表

10.2.3 迭代法描述

还是同一个列表（参见图 10.13），让我们利用迭代法将它排序。

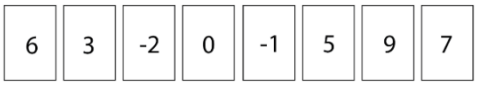


图10.13 列表

将列表中的元素看成一个个独立的子列表。迭代法的核心思想是利用 for 循环把这些子列表成对排序合并，最后组成一个列表整体。第一步，如图 10.14 所示，将长度为 1 的子列表排序合并成长度为 2 的子列表。和递归法一样，利用指针的方法排序。

注意：图中的箭头不代表指针。

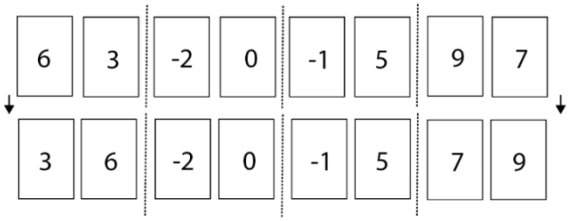


图10.14 第一步

第二步，如图 10.15 所示，将长度为 2 的子列表排序合并成长度为 4 的子列表。

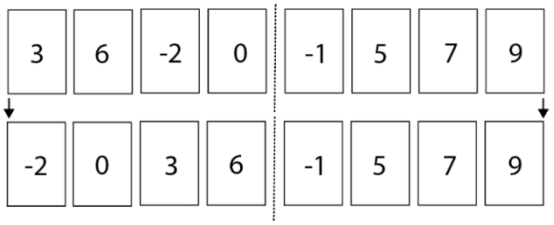


图10.15 第二步

最后，如图 10.16 所示，将长度为 4 的子列表排序合并成完整列表。

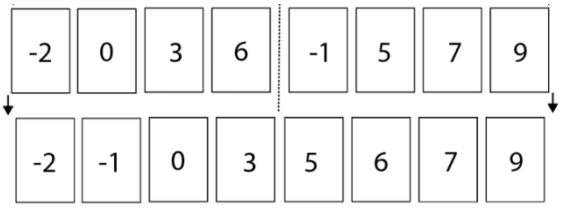


图10.16 第三步

递归法第一步排序左子列表，第二步排序右子列表，迭代法每一步都排序所有的子列表。两者的时间复杂度是一样的。

10.2.4　最终代码

代码 10.1 所示的是递归法排序算法的代码。

代码 10.1　归并排序——递归法排序代码

```
01  def mergeSort(myList):                          #传入参数 myList
02    if len(myList)<2:                             #如果 myList 只有一个元素，终止
03      return
04    cut = len(myList)//2                          #找到列表中点
05    listA = myList[:cut]                          #左子列表
06    listB = myList[cut:]                          #右子列表
07    mergeSort(listA)                              #把左子列表归并排序
08    mergeSort(listB)                              #把右子列表归并排序
09    pointerA = 0                                  #指针 A
10    pointerB = 0                                  #指针 B
11    counter=0                                     #定义一个 counter
12    while (pointerA<len(listA) and pointerB <len(listB)):  #如果两个指针都没走到头
13      if(listA[pointerA]<listB[pointerB]):        #比子列表元素大小
14        myList[counter]=listA[pointerA]           #小的元素替换 myList[counter]
15        pointerA+=1                               #移动指针
16      else:
17        myList[counter]=listB[pointerB]
18        pointerB+=1
19      counter+=1                                  #counter 加 1
20    while pointerA<len(listA):                    #如果指针 B 走到头但指针 A 没有
21      myList[counter]=listA[pointerA]             #指针 A 对应的元素替换 myList[counter]
22      pointerA+=1
23      counter+=1
24    while pointerB<len(listB):                    #如果指针 A 走到头但指针 B 没有
25      myList[counter]=listB[pointerB]             #指针 B 对应的元素替换 myList[counter]
26      pointerB+=1
27      counter+=1
28  listX = [2,3,4,1,3,0]
29  mergeSort(listX)
30  print(listX)
```

输出：

```
[0, 1, 2, 3, 3, 4]
```

代码 10.2 所示的是迭代法的代码。

代码 10.2　归并排序——迭代法代码

```
def mergeSort(myList):                                  #传入参数 myList
  length = len(myList)                                  #myList 的长度
  n = 1                                                 #n 为子数组长度，初始值为 1
  while n <length:                                      #如果子数组长度小于 myList 的长度
    for i in range(0,length,n*2):                       #利用 for 循环把子数组成对排序
      listA = myList[i:i+n]                             #listA 与 listB 为一对长度为 n 的子数组
      listB = myList[i+n:i+n*2]
      pointerA = 0                                      #利用指针将 listA 与 listB 排序
      pointerB = 0
       counter = i
      while (pointerA<len(listA) and pointerB <len(listB)):
        if(listA[pointerA]<listB[pointerB]):
          myList[counter]=listA[pointerA]
          pointerA+=1
        else:
          myList[counter]=listB[pointerB]
          pointerB+=1
        counter+=1
       while pointerA<len(listA):
        myList[counter]=listA[pointerA]
        pointerA+=1
        counter+=1
       while pointerB<len(listB):
        myList[counter]=listB[pointerB]
        pointerB+=1
        counter+=1
     n=n*2                                              #将 n 乘 2
listX = [-2,9,0,8,5,-4,8]
mergeSort(listX)
print(listX)
```

输出：

```
[-4, -2, 0, 5, 8, 8, 9]
```

可以看出两种方法的代码有一大部分是重复的，这是因为两者都用指针排序合并两个子列表，只不过一个利用递归，一个利用迭代。

10.3　连续子列表的最大和

我们的任务是在一个列表中找到连续子列表的最大和。列表中的数字可负可正，并且子列表不能为空。这个问题也是分治的一个经典问题。

10.3.1　解题思路

来看一个例子，如图 10.17 所示，目的是找到以下列表的最大子列表的和：

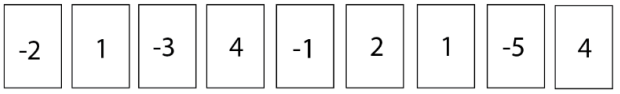


图10.17　列表

最大子列表有可能在左子列表、右子列表或左子列表与右子列表之间。我们需要做的是找到左子列表的最大子列表的和、右子列表的最大子列表的和、左子列表与右子列表之间的子列表的最大和，再进行比较。

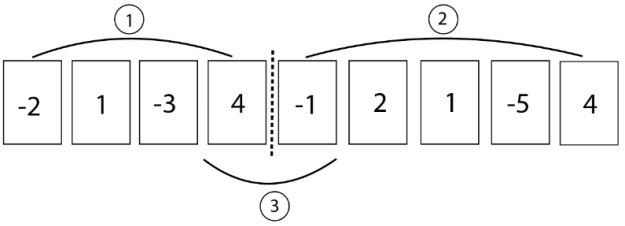


图10.18　三种可能

如何找到左子列表与右子列表的最大子列表的和呢？分治的想法是：让左子列表与右子列表的子列表回答这个问题就可以了，此时此刻不需要知道答案，只需要知道答案只有三种可能。

现在需要做的是找到第三种可能，也就是左子列表与右子列表之间的子列表的最大和。设一个中点，遍历中点左边的值，跟踪记录已遍历过的值的总和，取这些总和的最大值；遍历中点右边的值，跟踪记录已遍历过的值的总和，取这些总和的最大值。最后，左面的最大值加上右面的最大值加上中点值就是想要的值。通过上述方法，得到 6，也就是 4+(−1)+2+1。来看一下怎么用代码实现吧。

```
01    cut = len(nums)//2                    #设置中点位置，nums 为列表
02    leftMiddleSum=0                       #记录左面已遍历过的值的总和
03    maxL = 0                              #左面总和的最大值
04    rightMiddleSum=0                      #记录右面已遍历过的值的总和
05    maxR =0                               #右面总和的最大值
```

```
    for i in range(cut-1,-1,-1):                #遍历中点左面的元素
      leftMiddleSum+=nums[i]                    #更新 leftMiddleSum
      maxL = max(leftMiddleSum,maxL)            #比较判断是否更新 maxL
    for i in range(cut+1,len(nums),1):          #遍历中点右面的元素
      rightMiddleSum+=nums[i]                   #更新 rightMiddleSum
      maxR = max(rightMiddleSum,maxR)           #比较判断是否更新 maxR
     middleSum = maxL + nums[cut] + maxR        #第三种可能的值
```

现在来找左子列表的最大和。如图 10.19 所示，对待子列表的方式与对待列表的方式相同。还是有三个可能值，第一个与第二个可能值暂时不关心，让左子列表的子列表解决。找到第三个可能值就可以了，利用上述方法得到 2：(−2)+1+(−3)+4。

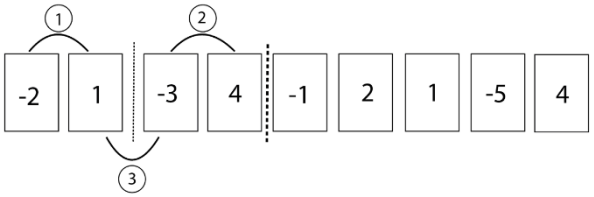


图10.19　左子列表的三种可能值

现在需要找到左子列表的左子列表的最大和。如图 10.20 所示，第一种可能是−2，第二种可能是 1，第三种可能是−1，返回三者的最大值：1。

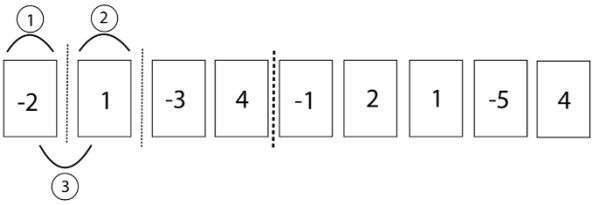


图10.20　左子列表的左子列表的三种可能值

同理，左子列表的右子列表的返回值是 4。如图 10.21 所示，左子列表的两个子列表返回对应的最大和，之前已经得出第三种可能值，所以现在可以进行比较。三种可能是：1、4、2，返回三者中的最大值：4。

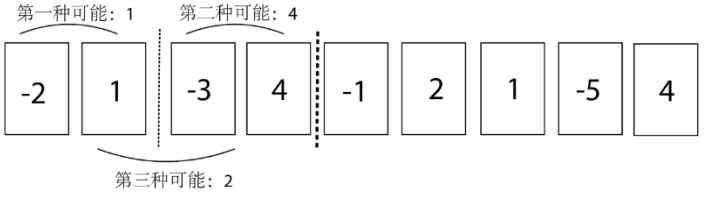


图10.21　得到左子列表的三种可能值

以同样的方式对待列表的右子列表，得到第二可能值：4。如图 10.22 所示，最大值（答案）是 6，所以输出 6。

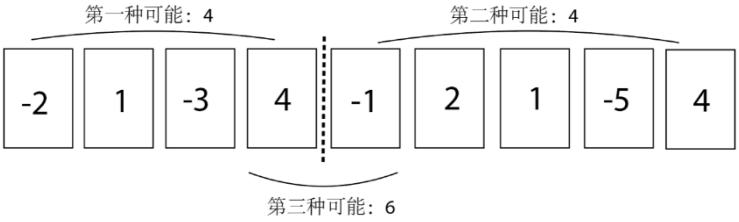


图10.22　得到列表的三种可能值

10.3.2　最终代码

找到连续子列表的最大和的代码如代码 10.3 所示。

代码 10.3　连续子列表的最大和

```
def maxSubArray(nums):                  #传入参数 nums 列表
  if nums==[]:                          #如果列表为空，终止
     return
  if len(nums)==1:                      #如果列表只有一个元素，输出此元素
     return nums[0]
  cut = len(nums)//2                    #设中点
  leftSum = maxSubArray(nums[:cut])  #分治：找到左子列表的最大和
  rightSum = maxSubArray(nums[cut:])#找到右子列表的最大和
  leftMiddleSum=0
   maxL = 0
   rightMiddleSum=0
   maxR =0
   for i in range(cut-1,-1,-1):       #找到左子列表与右子列表之间的子列表的最大和
     leftMiddleSum+=nums[i]
     maxL = max(leftMiddleSum,maxL)
   for i in range(cut+1,len(nums),1):
     rightMiddleSum+=nums[i]
     maxR = max(rightMiddleSum,maxR)
   return(max(leftSum,maxL+nums[cut]+maxR,rightSum))        #返回三个可能的最大值
maxSum = maxSubArray([-2,1,-3,4,-1,2,1,-5,4])               #示例
print(maxSum)
```

输出：

```
6
```

10.4 几何问题之凸包

简单地讲，凸包是正好包含所有点的凸多边形，可以将它想象成一个包围点集合的橡皮筋。如图 10.23 所示，之所以叫凸包正是因为这个“凸”多边形“包”围所有点。这一节的问题是怎样在点集合中找出凸包的顶点集。这一问题有不同的计算方式，常见的有 Graham 扫描法、Jarvis 步进法、快包法、分治法，本节讲解分治法。

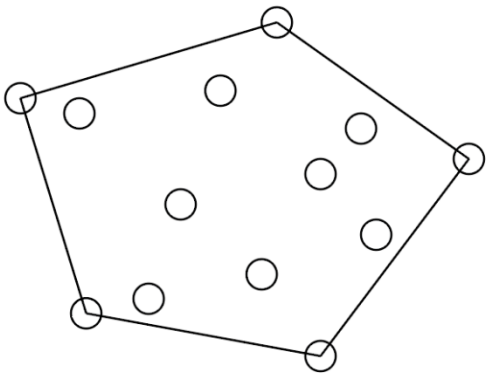


图10.23 凸包

10.4.1 问题求解

给予点集合，输出凸包顶点集。

```
Input : [(0, 0), (0, 4), (-4, 0), (5, 0), (0, -6), (1, 0)]            //点集合
Output : [(-4, 0), (5, 0), (0, -6), (0, 4)]                           //顶点集
```

分治法讲究“分而治之”。核心思想很好理解，难的是想出实施的具体步骤。怎么把大问题拆分成小问题并且确保两者的结构相同，是问题的关键。凸包问题的分治算法不唯一，这里讲解其中的一种。

首先，如图 10.24 所示，连接最左端与最右端的两点。命名线上方的区域为上包，线下方的区域为下包。同时，把这两点加入顶点集。

接下来，如图 10.25 所示，找到与直线距离最远的两点，与第一步中的两点相连，划分上包与下包。这里，把问题分解成了 4 个子问题。新找出的这两点肯定是凸点的顶点，所以把它们也加入顶点集。

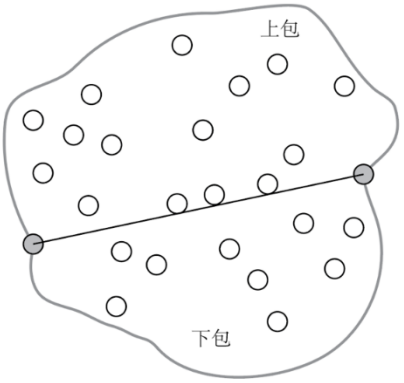


图10.24　第一步

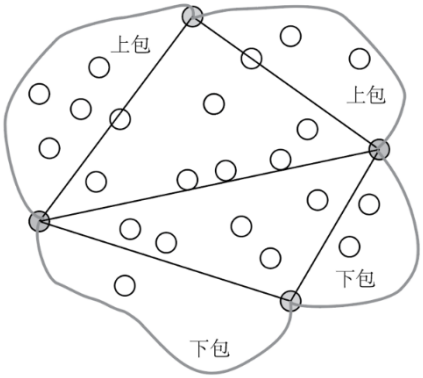


图10.25　第二步

寻找各包中与直线距离最远的点，连接，划分子上包与子下包，如图 10.26 所示。直到上包与下包为空，终止此循环，输出顶点集。注意左下包只包括一个点，所以没有必要划分子下包。图 10.27 展示了顶点集。

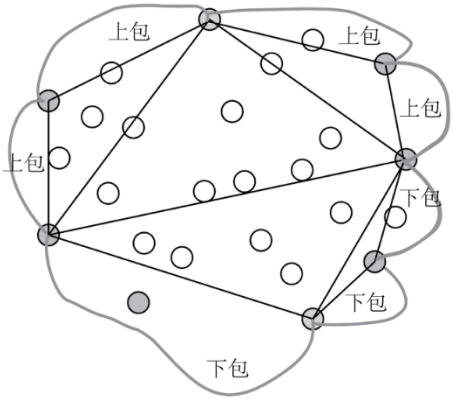


图10.26　第三步

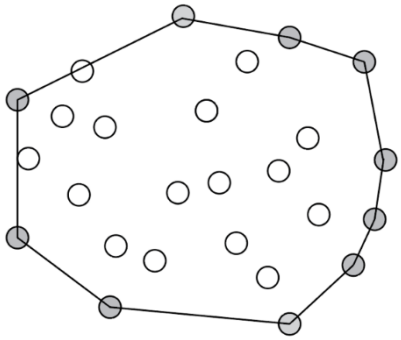


图10.27　输出顶点集

总结一下，首先分出上包与下包，然后找两个最远点连线，再次分出两个上包与两个下包。在此之后，上包递归分出两个子上包，下包递归分出两个子下包，直到包中节点数小于等于 1。每一次递归都把最远点加入顶点集。分治的思路在于每一次递归只找一个顶点，并把剩余顶点交给子问题，也就是子包去解决。

有了思路就容易写代码了。首先构建一个 lineHelper 方法。传入两点，lineHelper 函数输出直线 Ax+By+C=0 中的[A,B,C]。

```
01    def lineHelper(point1,point2):                    #传入两点，输出[A,B,C]
02      if (point1[0]-point2[0])!=0:                    #如果不为竖线
```

```
03        m = (point1[1]-point2[1])/(point1[0]-point2[0])
04        c = point1[1] - m*point1[0]
05        return [m,-1,c]
06      else:                                                    #竖线，比如 x=1
07        return [1, 0, point1[0]]
```

如果将[A,B,C]常数与一点的（x,y）值传入距离公式，即可得出此点与该直线之间的距离。

```
01    l = lineHelper(point1,point2)
02    distance = (l[0]*point[0]+l[1]*point[1]+l[2])/sqrt(l[0]*l[0] + l[1]*l[1])
```

接下来，声明一个全局变量 solution，也就是最后输出的顶点集。这里 solution 必须为全局变量，因为接下来的 helper 方法和 convexHull 方法都会对它进行更新。

```
solution = []
```

注意：在方法中引用全局变量 variable_name 需提前声明，global variable_name，否则变量 variable_name 自动被定义为局部变量。

为了不构建两个 helper 方法区别对待上包与下包，加入参数 upBool。upBool 为 True 代表上包，为 False 代表下包。

```
def helper(points,left_most,right_most,upBool):
```

helper 方法主要完成以下工作：

1. 利用 lineHelper 计算点 left_most 和 right_most 的直线公式。
2. 计算点集合中每点与直线的距离。
3. 如果为上包，把距离大于零的加入新集合，并找出最远点。
4. 如果为下包，把距离小于零的加入新集合，并找出最远点。
5. 再次调用 helper 方法，传入参数新集合与最远点。

下面来看一下代码和注释吧，如代码 10.4 所示。

10.4.2 最终代码

代码 10.4 几何问题之凸包

```
01    from math import sqrt
02    solution = []                                            #顶点集
03    def convexHull(points):
04      if len(points)<=3:                                     #少于三点
```

```
05        return points
06      global solution                                    #引用全局变量
07      points.sort(key=lambda x: x[0])                    #按 x 坐标排序
08      left most = points[0]                              #最左点
09      right most = points[len(points)-1]                 #最右点
10      solution.extend(left most,right most)              #加入顶点集合
11      helper(points,left most,right most,True)           #分治：上包
12      helper(points,left most,right most,False)          #分治：下包
13      return solution                                    #输出
14    #helper 方法，参数为：点集合，最左点，最右点，布尔值（上包为 True，下包为 False）
15    def helper(points,left most,right most,upBool):
16      global solution
18      if len(points) <= 1:                               #如果点数为 1 或 0
19        return               #注意：除第一次调用，点合集不包括最左点与最右点
20      l = lineHelper(left most,right most)               #直线
21      if upBool:                                         #如果为上包
22        up = []                                          #在直线之上的点集合
23        max_distance = 0
24        max_point = ()
25        for point in points:
26          distance = 0-(l[0]*point[0]+l[1]*point[1]+l[2])/sqrt(l[0]*l[0] + l[1]*l[1])
                                                           #与直线的距离
27          if distance >0:
28            up.append(point)
29            if distance>max_distance:
30               max_distance = distance
31               max_point = point
32        if max_point!=():
33           solution.append(max_point)                    #将最远点加入顶点集
34        helper(up,left_most, max_point,True)             #递归左上包
35        helper(up,max_point,right_most,True)             #递归右上包
36      else:                                              #如果为下包
37        down = []
38        min_distance = 0
39        min_point = ()
40        for point in points:
41          distance = 0-(l[0]*point[0]+l[1]*point[1]+l[2])/sqrt(l[0]*l[0] + l[1]*l[1])
42          if distance <0:
43            down.append(point)
44            if distance<min_distance:
45               min_distance = distance
46               min_point = point
47        if min_point!=():
```

```
48            solution.append(min point)
49          helper(down,left most,min point,False)      #递归左下包
50          helper(down,min point,right most,False)     #递归右下包
51
52      def lineHelper(point1,point2):              #传入两点，计算直线 Ax+Bx+C=0，输出[A,B,C]
54        if (point1[0]-point2[0])!=0:
56          m = (point1[1]-point2[1])/(point1[0]-point2[0])
57          c = point1[1] - m*point1[0]
59          return [m,-1,c]
60        else:
62          return [1, 0, point1[0]]
```

10.5 数学问题之多项式乘法

多项式乘法是一项基本的数学操作，在数学课上我们都学过怎样手动地把两个括号中的多项式展开。可以让计算机模拟人类，用同样的方式进行计算，但是这一节利用快速傅里叶变换（简称 FFT）创建一个更快捷的方法。FFT 采用分治思想降低时间复杂度。

问题为：给予两个多项式$A(x)$与$B(x)$，得出$C(x) = A(x)B(x)$。举例，$A(x) = 3 + 2x + 3x^2 + 4x^3$, $B(x) = 2 + x^2$, $C(x) = 6 + 4x + 9x^2 + 10x^3 + 3x^4 + 4x^5$。输入与输出为多项式的系数。

```
输入：[[3,2,3,4],[2,0,1]]
输出：[6,4,9,10,3,5]
```

10.5.1 问题求解

解决问题之前需要补充一些必要的数学知识。一般情况下，我们都用系数表示多项式，比如[6,4,9,10,3,4]代表$6 + 4x + 9x^2 + 10x^3 + 3x^4 + 4x^5$。但是，除了用系数表示，多项式还有一种表达方式叫作点值表示。

如果多项式的度数为 n，也就是说，多项式包含x^n项，则多项式可被 n+1 对点值表示，前提条件为点不重复。举例如下。

多项式：$A(x) = 3 + 2x + 3x^2 + 4x^3$

取 4 个点：$x = (0, 1, 2, 3)$

点值对：$(0, 3)$，$(1, 12)$，$(2, 51)$，$(3, 144)$

以上 4 点足以表示多项式$A(x)$，没有任何其他多项式拥有这 4 个点。

当两个多项式相乘时，只需相乘它们的点值对就可以得到结果多项式的点值表示。举例如下。

多项式：$A(x) = 3 + 2x + 3x^2 + 4x^3$，$B(x) = 2 + x^2$

取 6 个点：x=$(-2, -1, 0, 1, 2, 3)$

点值对：$A(x)$：(−2, −21), (−1, 0), (0, 3), (1, 12), (2, 51), (3, 144)

$B(x)$：(−2, 6), (−1, 3), (0, 2), (1, 3), (2, 6), (3, 11)

点值乘积：$C(x)$：(−2, −126), (−1, 0), (0, 6), (1, 36), (2, 306), (3, 1584)

需取 6 点，因为多项式 $C(x)$的度数为$2 + 3 = 5$。

知道了如何将系数表示转换为点值表示、如何做多项式点值表示的乘法，就可以开始学习 FFT 算法。FFT 算法可以做以下工作。

设结果多项式 $C(x)$的度数为 n。

- 找到单位的 n+1 次根，总共有 n+1 个。
- 通过分治快速计算 $A(x)$与 $B(x)$在这些单位根的值。
- 将 $A(x)$与 $B(x)$的点值相乘，得到 $C(x)$的点值表示。
- 将 $C(x)$的点值表示转换为系数表示。

FFT 的要点在于选值。如果只是随便选 n+1 个点，那就需要逐个计算这些点对应的值。但是可以利用单位根的特性，从而采取分治算法。让我们通过一个例子理解 FFT 算法以及单位根的特殊性。设

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

$A(x)$的系数表示为 A= $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$。

将 A 分成两部分，一部分为偶数系数，另一部分为奇数系数：

$$A_\text{even} = [a_0, a_2, a_4, a_6]$$
$$A_\text{odd} = [a_1, a_3, a_5, a_7]$$

以上两个系数表示代表的多项式为

$$A_\text{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$$
$$A_\text{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$$

那么，$A(x)$可表示为

$$A(x) = A_\text{even}(x^2) + xA_\text{odd}(x^2)$$
$$A(-x) = A_\text{even}(x^2) - xA_\text{odd}(x^2)$$

也就是说，只需计算 $A(x)$即可得到 $A(-x)$，毕竟两者唯一的不同在于加减法。取点时应该有效利用这个条件，令前半部分的点为后半部分的点的负数。这样的话，计算量直接减半。

比如，如果取 8 个点时，应取[−4, −3, −2, −1, 1, 2, 3, 4]而不是[1，2，3，4, 5, 6, 7, 8]。这样，只需计算 $A(-4)$、$A(-3)$、$A(-2)$、$A(-1)$，即可得到对应的 $A(4)$、$A(3)$、$A(2)$、$A(1)$。

但是记住，并不只分治一次，因为 A_even 和 A_odd 也将递归分解。因此，[−4, −3, −2, −1, 1, 2, 3, 4]只在第一次递归时有利。第二次递归时，传入 A_even 的点为[1, 4, 9, 16]，而不是理想中的[−1,−4,1,4]，因此第一次递归后没有再减少计算量。

只有选单位根为点才能达到目的。因为单位根的特性，每一次递归都能确保前半部分的点为后半部分的点的负数。举例如下。

单位的 8 次根为：1, ω, i, ω^3, −1, $-\omega$, −i, $-\omega^3$（ω为$2\sqrt{2}+2\sqrt{2}$i）。

点的平方：1, i, −1, −i

再次平方：1, −1

最后平方：1

来看一个实例：

$$A(x) = 2 + 3x - 4x^2 + 5x^3 - 6x^4 - 7x^5 + 8x^6 + 9x^7。$$

$$A(1) = A_even(1^2) + xA_odd(1^2)$$
$$A(\omega) = A_even(\omega^2) + xA_odd(\omega^2)$$
$$A(\mathrm{i}) = A_even(\mathrm{i}^2) + xA_odd(\mathrm{i}^2)$$
$$A(\omega^3) = A_even((\omega^3)^2) + xA_odd((\omega^3)^2)$$

在上一步的计算中，把 A_even(1^2), A_even(ω^2), ... , A_odd $((\omega^3)^2)$储存在列表中，以便直接计算值：

$$A(-1) = A_even(1^2) - xA_odd(1^2)$$
$$A(-\omega) = A_even(\omega^2) - xA_odd(\omega^2)$$
$$A(-\mathrm{i}) = A_even(\mathrm{i}^2) - xA_odd(\mathrm{i}^2)$$
$$A(-\omega^3) = A_even((\omega^3)^2) - xA_odd((\omega^3)^2)$$

又因为

$$A_even(1^2) = A_even(-\mathrm{i}^2),\quad A_even(\omega^2) = A_even(-(\omega^3)^2),$$
$$A_odd(1^2) = A_odd(-\mathrm{i}^2),\quad A_odd(\omega^2) = A_odd(-(\omega^3)^2),$$

所以，在第一步中，无须直接计算 $A_even(1^2)$，$A_even(\omega^2), \ldots, A_odd((\omega^3)^2)$的后半部分，而是在分治时采用同样的方法，将 $A_even_even(1^4)$、$A_even_even(\omega^4)$、$A_odd_odd(1^4)$，$A_odd_odd(\omega^4)$的值储存在列表中，方便计算后半部分。

到现在为止，介绍了如何把系数表示有效地转换为点值表示，下一步为把 $C(x)$的点值表示转换为系数表示。

这一步同样用 FFT 方法。之前用 n 个系数值求 n 个点值，现在反过来用 n 个点值求 n 个系数值。如图 10.28 与图 10.29 所示，除了单位根变为自己的倒数，其余保持不变。

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & & \omega^{n-1} \\ & \vdots & & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

图10.28　求点值

$$\frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & & \omega^{-(n-1)} \\ & \vdots & & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

图10.29　求系数

10.5.2　最终代码

```
from cmath import pi
from cmath import exp
import cmath
import numpy as np
#设 A(x)度数为 n-1，A 为多项式的系数表示，例如[3,4,0,0]，A 的长度为 2 的幂次方
def FFT(A,w):                          #w 为单位的 n 次本原单位根，比如，n=4 时 w=i
  length = len(A)
  if length==1:                        #如果 A 只有常数项，返回常数项系数
    return [A[0]]
  else:
    A_even =[]                         #在偶数位置的系数，例如[a_0,a_2，a_4]
    A_odd = []                         #在奇数位置的系数，例如[a_1,a_3，a_5]
    for i in range(0,length//2):
      A even.append(A[2*i])
      A odd.append(A[2*i+1])
    F even = FFT(A even, w**2)         #分治：递归计算 A even 与 A odd 的点值表示
    F_odd = FFT(A_odd,w**2)            #w 的平方
```

```
        x=1
        values = [None]*length                  #A 的点值表示
        for i in range(0,length//2):
          values[i]=F even[i] + x*F odd[i]      #上面讲到的公式
          values[i+length//2]=F even[i] - x*F odd[i]
          x=x*w
        return values

    def solver(A,B):                    #A、B 的系数表示，长度为各自的度数
                                        #length 为大于 len(A)+len(B)-1 的最小的 2 的幂次方
      length = len(A)+len(B)-1
      n  = 1
      while 2**n < length:
        n+=1
      length = 2**n
      A.extend([0]*(length-len(A)))   #填充 0，将 A、B 的长度扩展到 length
      B.extend([0]*(length-len(B)))

      w =  exp(2*pi*1j/length)        #计算 w，单位的 length 次本原单位根
      A_values = FFT(A,w)             #A、B 的点值表示，长度为 length
      B_values = FFT(B,w)
      C_values = [A_values[i]*B_values[i] for i in range(length)]        #C 的点值表示
                          #将 C 的点值表示转换为系数表示，FFT 传入 w 的倒数
      result = [int((x/length).real) for x in FFT(C_values,w**-1)]

      while result[-1] == 0:          #将 result 尾部不必要的 0 删除
        del result[-1]

      print(result)
```

输入：

```
solver([3,2,3,4],[2,0,1])
```

输出：

```
[6, 4, 9, 10, 3, 4]
```

FFT 算法要求多项式的系数表示长度为 2 的幂次方，因此 solver 方法用 0 扩展长度。